UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

PROYECTO FIN DE CARRERA

Optimización del consumo de energía en aplicaciones de redes de sensores inalámbricas

AUTOR: Daniel García Aubert
TUTOR: M. Soledad Escolar Díaz

Mayo, 2010

“Haz todo tan simple

como sea posible,

pero no más simple.”

Albert Einstein.

A Soledad Escolar, la directora de este proyecto.
Gracias por tu dedicación y paciencia infinita.

A Rubén. Tu amistad, lo mejor que me llevo
de mi paso por la Universidad.

A mis amigos, la OTL.
Compartiendo toda una vida.

Para mi musa Alexis, solo con tu sonrisa
la vida me parece mejor.

A Elena, Isabel, Sofía y Natalia mis hermanas.
Que son mis cuatro madres.
Un millón de gracias.

Y para Luis y Sylvette, mis padres. Espero
que estéis orgullosos del pequeño.
No os olvido.

Abstract

This End Career Project focuses on knowledge and use of wireless sensor networks (WSN) technology.

This project builds a system that allows monitoring the temperature of a building to determine its energy efficiency. On the other hand, it will develop a model to determine the consumption of a sensor node and estimate its life time within the network.

The system consists of three applications. The first one, which is installed into sensors nodes, will monitor the temperature of a building and send this information through the network to the gateway node. The second one will be installed into the gateway node and it will gather all the information circulating from the network and forward it to a base station. The last one, installed on the base station, will display and store all the information from the sensor network for further study.

The sensor nodes are devices that have energy constraints owing to its wireless nature. The model to be built reveals the consumption of a node running a certain application and estimate the battery life. Thanks to this model it is possible to compare the energy consumption behaviour of a node for different implementations of an application, in order to seek the maximum lifetime of a sensor node.

Resumen

Este Proyecto Fin de Carrera se centra en el conocimiento y uso de la tecnología de redes de sensores inalámbricas (Wireless Sensor Networks, WSN).

En este proyecto se construye un sistema que permite la monitorización de la temperatura de un edificio para determinar su eficiencia energética. Por otro lado, se desarrollará un modelo que permita conocer el consumo de un nodo sensor y estimar su tiempo de vida dentro de la red.

El sistema a construir consta de tres aplicaciones. La primera, instalada en los nodos sensores, monitorizará la temperatura ambiente de un edificio y enviará esta información a través de la red hasta el nodo gateway. La segunda aplicación, instalada en el nodo gateway, se encargará de obtener toda la información que circule por la red y reenviarla a una estación base. Y una tercera, instalada en la estación base, permitirá visualizar y almacenar toda la información proveniente de la red de sensores para su posterior estudio.

Los nodos sensores son dispositivos que soportan grandes restricciones energéticas debido a su naturaleza inalámbrica. El modelo a desarrollar permitirá conocer el consumo que realiza un nodo ejecutando una determinada aplicación y estimar la duración de las baterías. A partir de este modelo es posible comparar el comportamiento del consumo energético de un nodo según las distintas implementaciones de una aplicación, con el objetivo de buscar aquella maximice el tiempo de vida de un nodo sensor.

ÍNDICE GENERAL

ÍNDICE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1 Introducción

Cuando tengo que explicar en qué consiste este proyecto fin de carrera a amigos que no tienen nada que ver con el mundo de la ingeniería utilizo el siguiente recurso: "¿Has visto esa película en la que perseguían tornados para tirarle cientos de bolitas y así descubrir cómo son y cuál es su origen?". Si tengo suerte y la respuesta es sí, les contesto: "Pues en eso mismo consiste, pero sin tornados". El título de esa película, que pasó sin pena ni gloria por nuestras sobremesas de fin de semana es "Twister" de 1996. Esto, que hace 14 años era ciencia-ficción, hoy en día ya no lo es.

Por ejemplo, en un área extensa de difícil acceso que se encuentre en peligro por el cambio climático y los efectos de los gases invernadero, sería posible esparcir miles de nodos sensores para que analicen variables medioambientales como la temperatura o la humedad y tener una pequeña estación base con un ordenador conectado a Internet que pueda recoger dicha información. Mientras, un científico puede analizar esos datos en su ordenador. Ni siquiera sería necesario que el científico estuviese en dicha zona, podría estar en cualquier punto del globo terráqueo.

Las redes de sensores inalámbricas son un conjunto de nodos distribuidos en el entorno e interconectados entre sí formando una red con el objetivo de medir una variable física o ambiental. Esta información se transmite por la red y finalmente es recopilada por una estación base para su posterior estudio.

El poder de las WSN reside en su habilidad de organizar un gran número de dispositivos (nodos). Si se quiere introducir en la red del sistema uno o varios nodos estos se configuran por sí mismos creando una nueva topología en la red. Si por el contrario, un nodo o varios fallan, una nueva topología se establece y toda la red continuará enviando datos. Las WSN tienen la capacidad de adaptarse a los cambios que se produzcan en los lugares donde se encuentren. Los mecanismos de adaptación pueden responder a cambios en la topología de la red o también pueden causar un cambio del modo de operación de la red.

Las redes de sensores inalámbricas se han convertido en una tecnología que se encuentra en continua expansión, tanto a nivel de hardware, con la evolución de los diferentes dispositivos que componen una WSN, como a nivel de software, que dota de nuevas funcionalidades a estos dispositivos dando como resultado un sin fin de posibilidades. En sus inicios el desarrollo de las WSN fue motivado por las aplicaciones militares, como la vigilancia del campo de batalla. Ahora sus aplicaciones son muchas, tienen aplicación industrial y civil, incluida la vigilancia en procesos industriales, control del medio ambiente, la vigilancia del hábitat, la automatización del hogar, control del consumo energético en edificios públicos y privados, así como el control eficiente del tráfico.

Este tipo de redes están enfocadas a estar formadas por un gran número de nodos. A diferencia de las redes de móviles que deniegan el servicio cuando muchos teléfonos están activos en un área pequeña, la interconexión de los sensores inalámbricos se vuelve más fuerte cuantos más nodos componen el sistema. Una WSN puede crecer hasta cubrir un área extensa. Cada nodo tiene un rango de comunicación entre 50 y 100 metros.

El gran éxito de la WSN se debe en gran medida a ciertas características físicas que hacen que se convierta en una tecnología interesante para su estudio. A los nodos les son impuestos unos consumos de energía austeros. Esto se debe a la necesidad de que los nodos estén operativos el mayor tiempo posible para que el conjunto de la red pueda seguir cumpliendo su cometido. En lugares donde las fuentes de alimentación alternativa son prácticamente inexistentes se convierte en una necesidad primordial. Otra característica o restricción es el tamaño de los nodos, cada vez más necesario para la mayoría de las aplicaciones. Por ello, cada nueva generación de nodos presenta tamaños más reducidos.

Desde el punto de vista del software, se han desarrollado múltiples plataformas para este tipo de sistemas embebidos, que tienen en cuenta las características y las restricciones de los nodos. Estas plataformas han sido desarrolladas tanto en universidades como en empresas privadas y multinacionales obteniendo diferentes resultados, lo que ha influido en su difusión y posterior uso. La principal característica de estos sistemas es que presentan una

estructura modular que se adapta a la perfección en sistemas con restricciones de memoria. Por otro lado también se han desarrollado nuevos lenguajes de programación para este tipo de sistemas que típicamente siguen un enfoque orientado a eventos que, por sus características, precisan estos dispositivos.

Aunque queda fuera del alcance de este proyecto es interesante mencionar una nueva generación de WSN. Esta redes pueden incluir otro tipo de dispositivos denominados "actuadores", lo cual transforma las WSN a WSAN (Wireless Sensor and Actuator Network), del inglés Red de Sensores y Actuadores Inalámbricos. Los nodos sensores se encargan de registrar condiciones físicas o ambientales mientras que los nodos actuadores desarrollan las acciones apropiadas basadas en los datos recopilados. La inclusión de actuadores en redes de sensores inalámbricas permite adquirir una nueva dimensión, dotando de control, por ejemplo, en sistemas de climatización, de iluminación, de humedad, etc. Con el fin de ofrecer un mejor rendimiento y eficiencia energética en edificios.

La Figura 1 muestra un escenario de una WSAN donde los distintos componentes cooperan entre sí para acometer un objetivo común. Los actuadores estarán conectados a los dispositivos sobre los que realizarán cambios según las decisiones tomadas por la red en respuesta a los datos obtenidos por los nodos sensores.

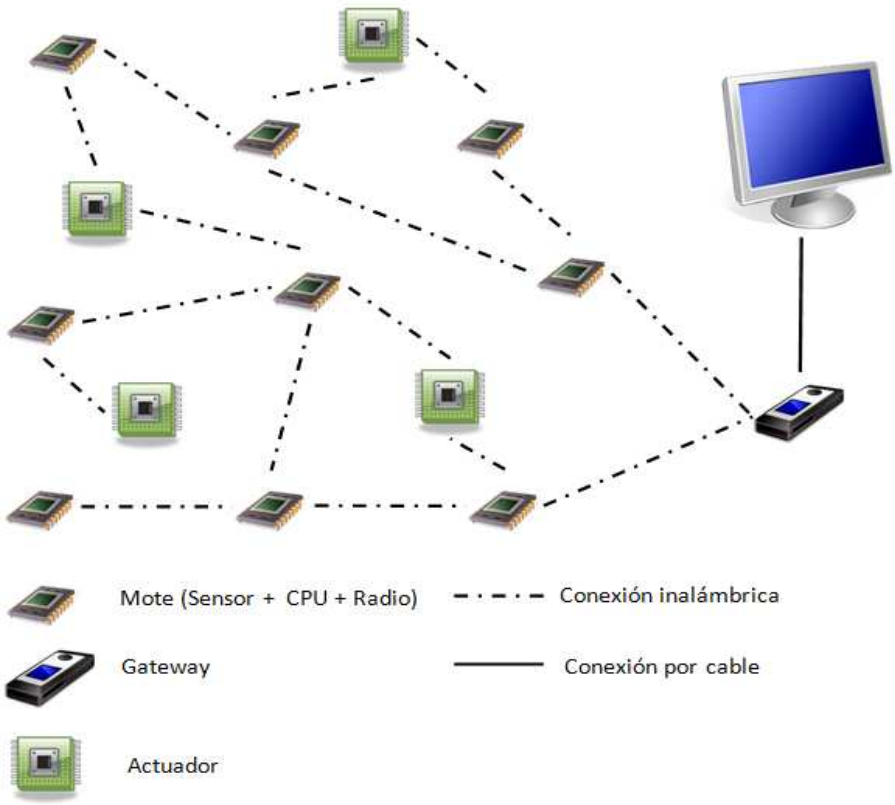


Figura 1: Escenario típico de una WSAN

1.1 Descripción del problema

El ámbito de este proyecto se encuentra dentro del marco de la monitorización de la temperatura de un edificio mediante una red de sensores inalámbrica para determinar el uso eficiente, o no, de los sistemas de calefacción/refrigeración de un edificio.

Para llevarlo a cabo, es preciso el conocimiento y uso de un conjunto de diferentes tecnologías. Se desea abarcar la tecnología Wireless Sensor Network desde sus orígenes hasta sus posibles líneas de trabajo en el futuro. Entender su filosofía, sus posibilidades y sus limitaciones. Además, conocer los dispositivos que componen una WSN. El software, desde el sistema operativo que gobiernan los nodos sensores hasta las herramientas de depuración y simulación. Comprender una nueva forma de programar orientada a eventos y las características del lenguaje de programación. Discernir los protocolos de comunicación entre los nodos sensores, ver las limitaciones físicas de la comunicación inalámbrica y como

superarlas. En definitiva, ver las posibilidades de una WSN y sus múltiples aplicaciones.

Por otro lado, uno de los mayores desafíos en el uso de las WSN es el consumo de energía y el tiempo de vida útil de la red. Los nodos sensores disponen de unas baterías con una capacidad de energía limitada y es por ello que el uso eficiente de los recursos del nodo se convierte en el mayor quebradero de cabeza en el desarrollo de aplicaciones para WSN. Se trata pues, de minimizar el consumo de energía de un nodo para maximizar su tiempo de vida en la red de sensores inalámbrica.

1.2 Motivación

Las WSN posibilitan múltiples aplicaciones, pero como se ha visto en el punto anterior, este proyecto se describe en el ámbito de la eficiencia energética en edificios. Se trata de un área que ha experimentado un crecimiento exponencial en los últimos años, tanto en los hogares como en edificios públicos o de oficinas. Más concretamente este proyecto estará enfocado en la monitorización de la temperatura en un edificio con el objetivo de obtener la información necesaria para poder determinar si el edificio optimiza o no el uso de los sistemas tanto de calefacción como de refrigeración. Para ello se estudiará el Proyecto de Real Decreto de Certificación Energética de Edificios.

El decreto aprobado por la Directiva 2002/91/CE (1) del Parlamento Europeo y del Consejo, de 16 de diciembre de 2002, relativo a la eficiencia energética de los edificios establece la obligación de poner a disposición de los compradores o usuarios de los edificios un certificado de eficiencia energética. Este certificado deberá incluir información objetiva sobre las características energéticas de los edificios de forma que se pueda valorar y comparar su eficiencia energética, con el fin de favorecer la demanda de edificios de alta eficiencia energética y las inversiones en ahorro de energía. Con el objetivo de aumentar la protección de los consumidores en cuanto a la información a suministrar en la compraventa y arrendamiento de viviendas, se fomentará la difusión de esta información; en particular, en el caso de las viviendas.

Con la elaboración de este proyecto, se pretende cubrir unos de los problemas que presenta la elaboración de un certificado energético para aquellos edificios que no hayan sido construidos bajo la directiva. Para ello se pretende utilizar la tecnología WSN para obtener la información necesaria con el fin de determinar si la utilización de los sistemas de calefacción o refrigeración es la adecuada en parámetros de eficiencia energética. Al monitorizar la temperatura cada cierto intervalo de tiempo, podremos evaluar si las temperaturas recibidas siguen un patrón de comportamiento eficiente o no. Para evaluar el resto de variables ambientales que determinen si se hace un uso eficiente de la energía, por ejemplo la luminosidad o el consumo de cada uno de los sistemas, se necesitan otro tipo de sensores, pero queda fuera del alcance de este proyecto.

Como ya se ha mencionado anteriormente, el tiempo de vida de un nodo sensor es uno de los mayores retos que tienen las WSN. En este proyecto se presenta un modelo de estimación del consumo energético de un nodo para ayudar a estimar el tiempo de vida del nodo dentro de la red de sensores. Para ello se ha tomado como base otros modelos, adaptados al hardware que se empleará en el desarrollo del proyecto. Una vez desarrollado el modelo se podrá determinar que estrategias son las mejores para llevar a cabo los objetivos planteados.

1.3 Solución propuesta y objetivos

En el uso de la tecnología WSN se ha de tener en cuenta aspectos, que por su naturaleza, son inamovibles y otros que por objetivos de este proyecto se hacen necesarios. A continuación se describirán los más importantes:

- Debe minimizarse los costes del proyecto. Aunque en la actualidad se están abaratando, el coste del hardware representa una de las principales barreras. El precio, por ejemplo, de un nodo sensor MicaZ ronda los 100 euros.
- La cobertura debe ser suficiente y por motivos de tolerancia a fallos será necesaria redundancia en el hardware.
- Se debe maximizar el tiempo de vida de la red de los dispositivos. En la actualidad, el ahorro en el consumo de energía de los dispositivos es una

cuestión crítica, pero a base de investigación se van solventando problemas.

- Su instalación ha de ser sencilla y su mantenimiento mínimo y remoto. No debe afectar a la instalación previa del edificio, no deben realizarse obras costosas para su implantación. El diseño de la aplicación debe maximizar el tiempo de vida de la red para que su mantenimiento se reduzca lo máximo posible además debe ser tolerante a fallos.
- La frecuencia de monitorización debe ser suficiente para llevar a cabo un estudio fiable.
- El sistema debe ser seguro. En comunicaciones inalámbricas la seguridad es un factor crítico. Aunque la encriptación del mensaje queda fuera del alcance de este proyecto.

El objetivo general de este proyecto es la construcción de un sistema capaz de monitorizar la temperatura de un edifico. Su solución necesita de la utilización de recursos hardware y software, que deberán proporcionar los siguientes objetivos:

- Obtener la monitorización de la temperatura en diversas estancias de un edificio.
- Por motivos de estudio del comportamiento del consumo energético de la red, controlar el estado de las baterías de los dispositivos.
- Por motivos de eficiencia en las comunicaciones establecer una estructura del mensaje lo más pequeña posible pero con toda la información necesaria.
- Optimizar el consumo de energía de toda la red obteniendo una frecuencia de monitorización.

Para llevar a cabo este proyecto, es necesario el estudio de la tecnología, tanto de su hardware como de su software. Además del manejo de las herramientas y elementos utilizados actualmente en el desarrollo de las WSN. Para el proyecto se ha utilizado como hardware la plataforma MicaZ, la placa programadora MIB520 (que conectada a un nodo MicaZ actuará como gateway) y la placa sensora MTS-310CB. Para el software se ha utilizado el sistema operativo

TinyOS en su versión 2.1. Además, NesC es el lenguaje de programación utilizado para el desarrollo de la aplicación. En los próximos capítulos del documento se ofrecerá información más detallada de todos los componentes usados de esta tecnología.

1.4 Estructura del documento

Este documento se estructura en nueve capítulos, una sección de bibliografía y dos anexos, en los que se explicará de manera detallada el trabajo realizado. A continuación, se realizará una breve descripción.

En el presente capítulo, se pretende ofrecer una visión global del contexto en el que se ha desarrollado el trabajo, la motivación y los objetivos del mismo, así como la mención de las herramientas principales utilizadas. Al final del capítulo, se muestra una relación de los términos, conceptos básicos y abreviaturas que pueden ser de gran utilidad para una mejor comprensión.

En el capítulo 2: *Estado de la cuestión*, se realizará un análisis sobre la tecnología a emplear, en el cual se establecerá una base necesaria sobre la que se asienta este proyecto.

El capítulo 3: *Entorno de Desarrollo*, aborda la tecnología hardware y software utilizado en el proyecto. Explica de manera detallada el nodo sensor utilizado y todos sus componentes (placa sensora, microprocesador, radio, etc.). Así como, el nodo gateway que tiene dos funciones: pasarela de información entre la red de sensores y la estación base y de placa programadora de los nodos sensores. Además, se explicarán todos los elementos software de los que se compone una WSN, tales como, sistema operativo, lenguajes de programación y simuladores.

A partir del capítulo 4: *Análisis del problema*, se afronta el estudio del problema al que nos enfrentamos. De manera más concreta, se centra en el análisis de los requisitos exigidos por el sistema a desarrollar, identificando de forma clara e individual cada uno de ellos.

El capítulo 5: *Diseño de la WSN*, aborda la solución para la obtención de los objetivos marcados al inicio. Se explican de manera detallada las decisiones de diseño tomadas.

En el capítulo 6: *Implementación de la WSN*, donde se explica en detalle la implementación en el lenguaje de programación seleccionado y que solucionan las especificaciones del diseño.

En el capítulo 7: *Modelo de consumo de nodo MicaZ*, se detalla el modelo de estimación del consumo de energía de un nodo MicaZ. A partir de la simulación se obtienen datos fiables para estimar el consumo que realiza un nodo ejecutando una aplicación, para luego poder calcular el tiempo de vida esperado del nodo sensor.

En el capítulo 8: *Evaluación*, se presentan una serie de pruebas para proporcionar un mejor entendimiento del sistema construido. Estas pruebas consisten en mostrar al lector cómo el uso de políticas de ahorro de energía y los distintos protocolos de enrutamiento pueden maximizar el tiempo de vida de la red y favorecer el desarrollo y el empleo de la tecnología WSN.

Consiste en la creación de tres escenarios de pruebas. Por una parte, se ejecutará la aplicación en un entorno de simulación y por otra sobre un entorno real.

En el capítulo 9: *Conclusiones*, trata de mostrar una visión sobre el trabajo realizado, evaluando la consecución de los objetivos. Además, se presentan los gastos inherentes al desarrollo del proyecto y se proponen una serie de mejoras y posibles líneas de trabajo futuras.

Se adjunta la bibliografía utilizada para este proyecto y dos anexos como material complementario, con el objetivo de que sirvan como ayuda para una mejor comprensión del trabajo realizado. Anexo A Avrora: muestra la instalación del simulador y las pautas para su ejecución. Anexo B Cálculos de los modelos energéticos: muestra en detalle los cálculos obtenidos de los modelos energéticos para cada escenario.

1.5 Terminología y conceptos básicos

A

Actuador

Son aquellos elementos que pueden provocar un efecto sobre un proceso automatizado. Los actuadores son dispositivos capaces de generar una fuerza a partir de líquidos, de energía eléctrica y gaseosa. El actuador recibe la orden de un regulador o controlador y da una salida necesaria para activar a un elemento final de control como lo son las válvulas. Existen tres tipos de actuadores:

- Hidráulicos.
- Neumáticos.
- Eléctricos.

B

Bluetooth

Es una especificación industrial para Redes Inalámbricas de Área Personal (WPANs) que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia segura y globalmente libre (2.4 GHz). Los principales objetivos que se pretende conseguir con esta norma son:

- Facilitar las comunicaciones entre equipos móviles y fijos.
- Eliminar cables y conectores entre éstos.
- Ofrecer la posibilidad de crear pequeñas redes inalámbricas y facilitar la sincronización de datos entre nuestros equipos personales.

C

CPU

Abreviatura de Central Processing Unit (Unidad Central de Proceso), se pronuncia como letras separadas. La CPU es el cerebro del ordenador. A veces es referido simplemente como el procesador o procesador central, la CPU es donde se producen la mayoría de los cálculos. En términos de potencia del ordenador, la CPU es el elemento más importante de un sistema informático.

D

Domótica

Se trata de un conjunto de sistemas capaces de automatizar una vivienda. Aporta servicios de gestión energética, seguridad, bienestar y comunicación, y pueden estar integrados por medio de redes de comunicación (interiores y exteriores), cableadas o inalámbricas, y cuyo control goza de cierta ubicuidad, desde dentro y fuera del hogar. Se podría definir como la *integración de la tecnología en el diseño inteligente de un recinto*.

E

Enrutamiento

También conocido como *encaminamiento*, se trata de la función de buscar un camino entre todos los posibles en una red de paquetes cuyas topologías poseen una gran conectividad. Dado que se trata de encontrar la mejor ruta posible, lo primero será definir qué se entiende por mejor ruta y en consecuencia cuál es la métrica que se debe utilizar para medirla.

ERS

La *Especificación de Requisitos Software* es una completa descripción del comportamiento del sistema que se quiere desarrollar. Además, describe las interacciones que tendrán los usuarios con el software.

F

FIFO

Es el acrónimo inglés de *First In, First Out* (primero en entrar, primero en salir). Un sinónimo de FIFO es FCFS, acrónimo inglés de *First Come First Served* (primero en llegar, primero en ser servido). Es un método utilizado en estructuras de datos, contabilidad de costes y teoría de colas.

G

Gateway

Pasarela que puede permitir la interconexión entre la red de sensores y una estación base o computador que suele disponer de una conexión directa a una red TCP/IP.

GNU GPL

La Licencia Pública General de GNU o más conocida por su nombre en inglés GNU General Public License o simplemente su acrónimo del inglés GNU GPL, es una licencia creada por la *Free Software Foundation* en 1989 (la primera versión), y está orientada principalmente a proteger la libre distribución, modificación y uso de software. Su propósito es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios.

I

Internet

Conjunto de redes de ordenadores creada a partir de redes de menor tamaño. Es la red global que utiliza el protocolo TCP/IP para proporcionar comunicaciones de ámbito mundial a hogares, negocios, escuelas y gobiernos.

M

Mote

Se denomina mote (o nodo sensor) al dispositivo con sensores y capacidad para realizar mediciones y transmitirlas, que colabora con otros para formar una WSN.

R

RS-232

También conocido como *Electronic Industries Alliance*, es una interfaz que designa una norma para el intercambio serie de datos binarios. Consiste en un conector de 25 pines (aunque es normal encontrar la versión de 9 pines), más barato e incluso más extendido para cierto tipo de periféricos (como el ratón serie del PC).

S

Sensor

Dispositivo cuyo fin es recibir estímulos y transformarlos en información, permitiendo así medir fenómenos físicos.

T

TCP/IP

Transmision Control Protocol/Internet Protocol, es el protocolo estándar de comunicaciones en red, utilizado para conectar sistemas informáticos a través de Internet.

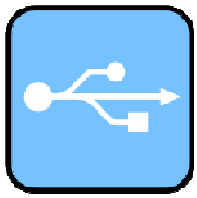
U

UML

Lenguaje Unificado de Modelado (*UML*, por sus siglas en inglés, *Unified Modelling Language*) es el lenguaje de modelado de sistemas de software más

conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema.

USB



El *Universal Serial Bus* (bus universal en serie) o Conductor Universal en Serie, es un puerto de comunicaciones que sirve para conectar periféricos a una computadora. Fue creado en 1996 por siete empresas: IBM, Intel, Northern Telecom, Compaq, Microsoft, Digital Equipment Corporation y NEC. Los dispositivos USB se clasifican en cuatro tipos según su velocidad de transferencia de datos:

- Baja velocidad (1.0): tasa de transferencia de hasta 1.5 Mbps (192 KB/s). Utilizado en su mayor parte por dispositivos de interfaz humana (*Human interface device*, en inglés) como los teclados, los ratones y los joysticks.
- Velocidad completa (1.1): tasa de transferencia de hasta 12 Mbps (1.5 MB/s). Ésta fue la más rápida antes de la especificación USB 2.0, y muchos dispositivos fabricados en la actualidad trabajan a esta velocidad. Estos dispositivos dividen el ancho de banda de la conexión USB entre ellos, basados en un algoritmo de búferes FIFO.
- Alta velocidad (2.0): tasa de transferencia de hasta 480 Mbps (60 MB/s).
- Super velocidad (3.0): actualmente en fase experimental y con tasa de transferencia de hasta 4.8 Gbps (600 MB/s). Esta especificación será lanzada a mediados de 2009 por Intel. La velocidad del bus será diez veces más rápida que la del USB 2.0, debido a que han incluido 5 conectores extra, desechando el conector de fibra óptica propuesto inicialmente y será compatible con los estándares anteriores. Se espera que los productos fabricados con esta tecnología lleguen al consumidor en 2010 o 2011.

W

Wi-Fi

Se trata de una de las tecnologías de comunicación inalámbrica mediante ondas más utilizada hoy en día. WIFI, también llamada WLAN (W*ireless LAN*, red inalámbrica) o estándar IEEE 802.11. WIFI no es una abreviatura de *Wireless Fidelity*, simplemente es un nombre comercial. En la actualidad podemos encontrarnos con dos tipos de comunicación WIFI:

- 802.11b, que emite a 11 Mb/seg.
- 802.11g, más rápida, a 54 MB/seg.

De hecho, su velocidad y alcance (unos 100-150 metros en hardware asequible) lo convierten en una fórmula perfecta para el acceso a internet sin cables.

WLAN

En inglés *Wireless Local Area Network* (*Red Inalámbrica de Área Local)*, es un sistema de comunicación de datos inalámbrico flexible, muy utilizado como alternativa a las redes LAN (Local Area Network, Red de Área Local) cableadas o como extensión de éstas. Utiliza tecnología de radiofrecuencia que permite mayor movilidad a los usuarios al minimizar las conexiones cableadas. Las WLAN van adquiriendo importancia en muchos campos, como almacenes o para manufactura, en los que se transmite la información en tiempo real a una terminal central. También son muy populares en los hogares para compartir el acceso a Internet entre varias computadoras.

WPAN

Wireless Personal Area Networks (*Red Inalámbrica de Área Personal* o *Red de área personal* o *Personal Area Network)* es una red de computadoras para la comunicación entre distintos dispositivos (tanto computadoras, puntos de acceso a internet, teléfonos celulares, PDA, dispositivos de audio, impresoras) cercanos al punto de acceso. Estas redes normalmente son de unos pocos metros y para uso personal, así como fuera de ella.

WSN

Wireless Sensor Network o Red de Sensores Inalámbrica, es una red basada en pequeños dispositivos dotados de sensores y capacidad para realizar mediciones, que cooperan entre sí para monitorizar un fenómeno.

Z

ZigBee

ZigBee es el nombre de la especificación de un conjunto de protocolos de alto nivel de comunicación inalámbrica para su utilización con radios digitales de bajo consumo, basada en el estándar IEEE 802.15.4 de redes inalámbricas de área personal (*Wireless Personal Area Network,* WPAN). Su objetivo son las aplicaciones que requieren comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías.

Capítulo 2 Estado de la cuestión

En este capítulo se presenta el estado de la cuestión de la tecnología *Wireless Sensor Networks*, que, como veremos más adelante, en los últimos años ha evolucionado de manera exponencial. Se describirán los aspectos más relevantes de la tecnología hardware y software así como los protocolos de comunicación utilizados en esta tecnología.

Como se mencionó en el capítulo anterior, una WSN es, "un conjunto de nodos distribuidos por el medio e interconectados entre sí formando una red, con el objetivo de medir una variable física o ambiental. Esta información se transmite por la red y finalmente es recopilada por una estación base para su posterior estudio". Estos nodos colaboran entre sí para acometer una tarea común, normalmente recolección de datos en múltiples aplicaciones de distintos campos tales como entornos industriales, domótica, entornos militares o la medición de diferentes parámetros medioambientales.

2.1. Historia de las WSN

Los sensores electrónicos se han utilizado desde hace mucho tiempo para registrar fenómenos físicos como la temperatura, el movimiento, la presencia de gases, los campos magnéticos, entre muchas otras aplicaciones. El uso de los sensores tiene muchas ventajas, entre ellas podemos destacar la detección oportuna de situaciones que representen un riesgo potencial. Sin embargo, hasta hace pocos años, eran soluciones en donde los sensores se conectaban por cable hasta un punto central de procesamiento y almacenamiento. La utilización de medios guiados limita las múltiples aplicaciones de una red de sensores.

Como casi todo en tecnología la evolución de redes de sensores inalámbricas tienen su origen en iniciativas militares, por lo cual, no hay información precisa sobre sus inicios. La investigación en redes de sensores comenzó en la década de 1980 con el proyecto *Distributed Sensor Networks (DSN)* de la agencia militar de investigación avanzada de Estados Unidos, en inglés,

Defense Advanced Research Projects Agency (DARPA). Además se sabe que una de las primeras redes de sensores es el sistema S*ound Surveillance System (SOSUS),* una red de boyas sumergidas instaladas en las costas de los Estados Unidos durante la Guerra Fría para detectar submarinos.

Como ya se ha mencionado anteriormente el tamaño de los nodos o motes es clave. En 1998 surgió el proyecto *Smart Dust* que explora las posibilidades y limitaciones de la nano-electrónica y la micro-fabricación para determinar como un dispositivo sensorial autónomo puede ser empaquetado en una "mota" de un milímetro cúbico, o menos, para formar la base de una red de sensores integrada que pueda ser masivamente distribuida. La idea fue concebida por el Dr. Kristofer Pister (Universidad de California, Berkeley). Su propuesta fue la de construir un dispositivo con un sensor, un sistema de comunicación, y una pequeña computadora, todo integrado en una sola plataforma.

La Figura 2 muestra un ejemplo de una WSN donde la red de sensores inalámbrica se encuentra conectada a una estación base donde se almacenan los datos obtenidos. A su vez la estación base se encuentra conectada a internet, para que cualquier ordenador pueda consultar la información de la WSN desde cualquier parte del mundo.

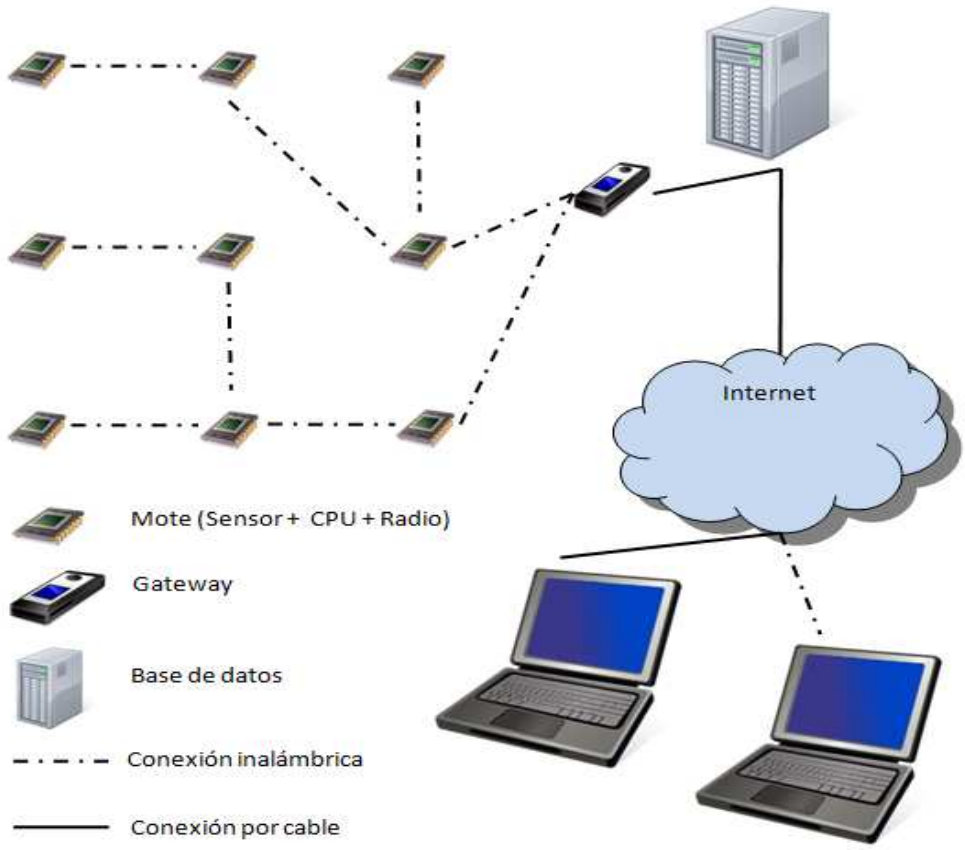


Figura 2: Escenario típico de una WSN

Las redes de sensores están siendo una de las batallas en el campo de investigación tanto en hardware como en software. Instituciones tan importantes como la NASA, que ha ido integrando a sus proyectos de exploración de la superficie de Marte redes inalámbricas de sensores autosuficientes esféricos. Con el objetivo de recabar una cantidad inmensa de datos para su posterior estudio.

Por tanto las redes de sensores inalámbricas tienen una historia breve, apenas dos décadas, pero a la vez muy intensa. Su evolución es exponencial y los últimos años han surgido multitud de proyectos. En el ámbito empresarial muchas empresas multinacionales están desarrollando sus propias plataformas de redes de sensores, un ejemplo claro es el proyecto *Sun Spot* de *Sun Microsystems, Inc* (2).

2.2 Componentes de una WSN

A continuación se describen en detalle los elementos que forman parte de una red de sensores inalámbrica.

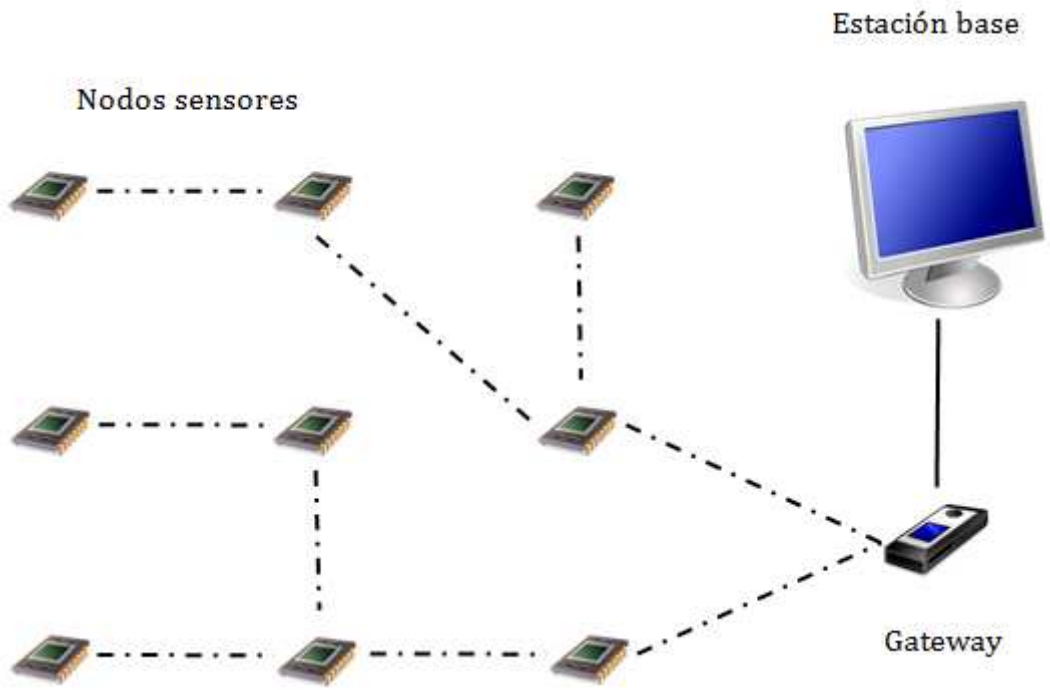


Figura 3: Componentes hardware de una WSN

2.2.1 Nodo sensor

Un nodo sensor, o mote, es un dispositivo compuesto por un microprocesador con memoria, uno o varios sensores, radio de baja potencia y una batería. Los nodos pueden incluir una gran variedad de sensores, estos pueden ser de temperatura, humedad, luminosidad, presión, detectores de gases (CO, CO2, CH4, etc.), de campos magnéticos, detectores de sonido y movimiento, etc. Estos sensores deben conectarse de forma sencilla al nodo sensor. Un nodo sensor integra, al menos, los siguientes componentes:

- Radio: existen varias alternativas, siendo la IEEE 802.15.4 (3) y la Zigbee (4) las más utilizadas por el amplio respaldo que cuentan de la industria a nivel mundial. Emplean diversas bandas de frecuencia de radio:
    - 2.4 GHz con una velocidad de transmisión de 250 Kbps,
    - 915 MHz (banda americana) con una velocidad de transmisión de 40 Kbps.
    - 868 MHz (banda europea) con una velocidad de transmisión de 40 Kbps.

- Microprocesador: su velocidad suele ser de varios Megahertzios y tiene una RAM y ROM reducida (4KB de RAM y 128 KB de EEPROM).
- Batería: generalmente son pilas de Níquel-Cadmio o batería de Litio, en muchos casos se utilizan pilas convencionales (2xAA) con capacidad aproximada de 2000-3000mAh. En la actualidad existen nodos sensores alimentados por baterías de litio que son recargados con energía alternativa a través de la luz e incluso a través de las vibraciones, por ejemplo, el nodo ekö de Crossbow Technology Inc (5).

La Figura 4 muestra un esquema de componentes de un nodo sensor típico.

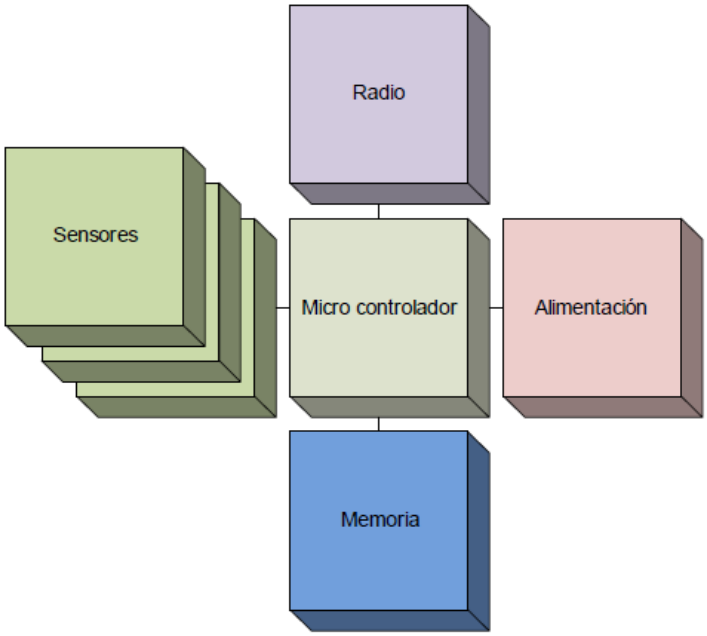


Figura 4: Componentes de un nodo sensor

En la actualidad existen multitud de plataformas de nodos sensores de los distintos fabricantes, varían en cuanto a su tamaño, consumo de energía, capacidad de proceso y almacenamiento. A continuación se muestra una lista de nodos sensores de las plataformas más comunes en el desarrollo de aplicaciones de redes de sensores inalámbricas.

2.2.1.1 BTNode3

El BTnode es un nodo sensor versátil basado en una radio Bluetooth, una radio de baja potencia y un microcontrolador. Fabricado por ETH Zúrich (Suiza) está diseñado como plataforma de demostración y creación de prototipos para la investigación en telefonía móvil, redes móviles Ad-Hoc (MANET) y redes de sensores (WSNs). La radio es la misma que en la Mote Mica2 de Berkeley.

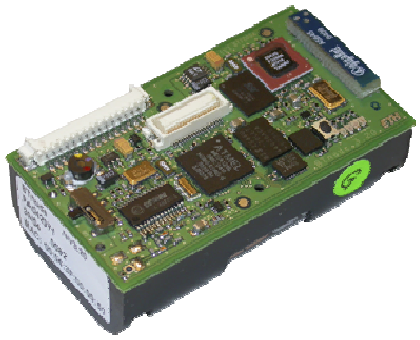


Figura 5: Nodo BTnode3 de ETH Zúrich

2.2.1.2 EyesIFX

Desarrollado por Infineon (6) en colaboración con el proyecto EYES de redes inalámbricas de sensores. Su microprocesador pertenece a la familia MSP430 producidos por Texas Instruments (7). La plataforma integra sensores de temperatura y de luz, además ofrece un soporte completo de software proporcionado por TinyOS de TU Berlín.

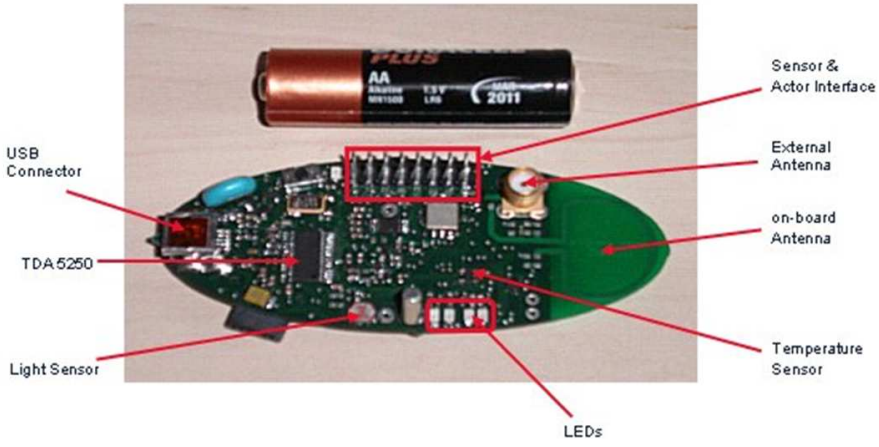


Figura 6: Nodo EyesIFX de Infineon

2.2.1.3 Imote2

El Imote2 es un avanzado nodo sensor inalámbrico fabricado por Crossbow Technology Inc. Está construido alrededor de la CPU XScale PXA271 de baja

potencia y también integra una radio IEEE 802.15.4. El diseño es modular y ampliable con conectores para tarjetas de expansión tanto en el lado superior como en el inferior.

Figura 7: Nodo Imote2 de Crossbow Technology Inc

2.2.1.4 Iris

El IRIS es un nodo sensor fabricado por Crossbow Technology Inc, está diseñado específicamente para sistemas embebidos. Cuenta con una radio de 2,4 GHz (IEEE 802.15.4) de baja potencia con una capacidad de transmisión de 250 Kbps. Además tiene conectores de expansión para sensores de luz, temperatura, humedad, presión barométrica, etc. El mote IRIS tiene nuevas funcionalidades que mejoran la funcionalidad general de los productos de redes de sensores inalámbricos.

Figura 8: Nodo Iris de Crossbow Technology Inc

2.2.1.5 Mica2

El mote Mica2 (Crossbow Technology Inc) pertenece a la tercera generación de módulos para redes de sensores. El nodo Mica2 tiene nuevas mejoras sobre el

original mote MICA. Cuenta con un transmisor-receptor multicanal (868/916 MHz). Cada nodo tiene capacidad para actuar como enrutador en las comunicaciones inalámbricas, además cuenta conectores de expansión para todo tipo de sensores (luz, temperatura, humedad, etc.).

Figura 9: Nodo Mica2 de Crossbow Technology Inc

2.2.1.6 Mica2Dot

El nodo Mica2Dot (Crossbow Technology Inc) también pertenece a la tercera generación de nodos sensores. Mica2Dot es similar a Mica2, a excepción de su tamaño que es una cuarta parte y en forma de moneda. Trae integrado un sensor de temperatura, LEDs y un monitor de batería. A destacar su radio multicanal (868/916 MHz, 433/315 MHz).

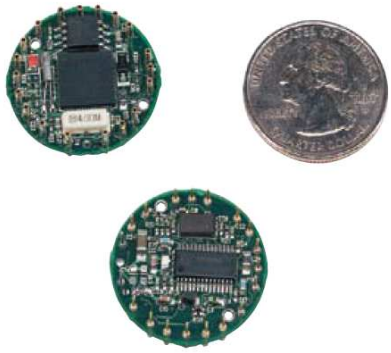


Figura 10: Nodo Mica2Dot de Crossbow Technology Inc

2.2.1.7 MicaZ

El mote MicaZ (Crossbow Technology Inc), utilizado en este proyecto, pertenece a la misma familia de nodos sensores que Mica, Mica2 y Mica2Dot. Todos ellos ofrecen características similares. En el siguiente capítulo se detalla en profundidad.

Figura 11: Nodo MicaZ de Crossbow Technology Inc

2.2.1.8 TelosB

TelosB (TPR2400), producido por Crossbow Technology Inc, es una plataforma de código abierto diseñado para permitir la experimentación en la comunidad de investigación. El nodo sensor trae lo esencial para todo tipo de estudios en una sola plataforma, incluyendo, programación vía USB, radio IEEE 802.15.4 integrada, ratio de transmisión de 250 Kbps, CPU 2.4 GHz con 10 Kb de RAM, bajo consumo, flash 1 Mb para almacenamiento de datos.

Figura 12: Nodo TelosB de Crossbow Technology Inc

2.2.1.9 TinyNode

El TinyNode es un módulo construido específicamente para funcionar bajo el sistema operativo TinyOS. Fabricado por la empresa Suiza Shockfish (8). Su diseño está basado para funcionar a potencia ultra baja, lleva incorporado un sensor de temperatura, además se puede integrar una amplia gama de sensores y actuadores. Utiliza el microprocesador MSP430 de Texas Instruments con radio multicanal de baja potencia.

Figura 13: Nodo TinyNode de Shockfish

2.2.1.10 Eko

Este modelo de última generación es un dispositivo robusto, de mayor tamaño, diseñado para estar al aire libre y alimentado por energía solar. Cada nodo puede alojar hasta 4 diferentes tipos de sensores. El nodo eko (Crossbow Technology Inc) integra la plataforma Iris con baterías recargables y una célula solar. Los nodos vienen pre-programados y pre-configurados para formar una red y requieren aproximadamente 1-2 horas por día de exposición al sol para mantener sus baterías cargadas. La carcasa cuenta con un soporte de metal en la parte trasera para que pueda ser fijado con total seguridad.

Figura 14: Nodo Eko de Crossbow Technology Inc

2.2.2 Nodo gateway

El gateway es un dispositivo que permite la conexión entre la red inalámbrica y la estación base. Otra de las funciones fundamentales de gateway es

la coordinación de la red de sensores. Sirve como pasarela entre los protocolos de comunicación ZigBee y TCP/IP o el puerto serie del PC al cuál se encuentra físicamente conectado. El gateway es un elemento crítico de la red de sensores inalámbricos ya que por él pasa todo el tráfico de la red y es en él donde puede producirse un cuello de botella. Es por ello que el gateway cuenta con mejores prestaciones que un nodo sensor y normalmente tiene alimentación por red eléctrica en lugar de baterías.

Existen multitud de gateways de distintos fabricantes, varían en cuanto a su tamaño, consumo de energía, capacidad de proceso y almacenamiento. A continuación se muestra una lista de gateways de las plataformas más comunes en el desarrollo de aplicaciones de redes de sensores inalámbricas.

2.2.2.1 MIB510

El MIB510 (por Crossbow Technology Inc) permite que los datos de la red de sensores lleguen a la estación base, así como otras plataformas de ordenador estándar. Este modelo puede usarse con redes de sensores que usen plataformas Iris/MicaZ/Mica2. Además de actuar como gateway, el MIB510 también proporciona una interfaz RS-232 de programación de nodos sensores.

Figura 15: Gateway MIB510 de Crossbow Technology Inc

2.2.2.2 MIB520

El MIB520CB (Crossbow Technology Inc) ofrece conectividad para las plataformas Iris, Mica2 y MicaZ. Además de la transferencia de datos, la MIB520CB también proporciona una interfaz de programación de nodos sensores vía USB. Este modelo es usado en este proyecto. Más adelante en el Capítulo 3 se detalla en profundidad sus características.

Figura 16: Gateway MIB520 de Crossbow Technology Inc

2.2.2.3 MIB600

El MIB600 (Crossbow Technology Inc) proporciona conectividad Ethernet (10/100 Base-T) a las plataformas Iris/MicaZ/Mica2 de motes de comunicación y en la programación del sistema. El gateway MIB600 permite el acceso remoto a los datos de los sensores de la red a través de TCP/IP. El modelo MIB600 se conecta directamente a una red 10 Base-T LAN como cualquier otro dispositivo de red.

Figura 17: Gateway MIB600 de Crossbow Technology Inc

2.2.2.4 NetBridge

Netbridge Stargate (Crossbow Technology Inc) es un gateway de nueva generación. Su objetivo es conectar los nodos Crossbow a una red Ethernet existente. Se basa en el procesador Intel XScale IXP420 corriendo a 266MHz. Incluye una conexión de cable Ethernet y dos puertos USB 2.0. El dispositivo también está equipado con 8 MB de memoria flash para programas, 32MB de RAM y 2GB USB 2.0 de disco del sistema.

Figura 18: Gateway Stargate Netbridge de Crossbow Technology Inc

Stargate Netbridge corre bajo el sistema operativo Linux Debian. Tiene preinstalado la gestión de la Red de Sensores y visualización de paquetes de datos de la propia red, además de los programas Xserve y MoteExplorer. Stargate Netbridge es un dispositivo totalmente plug-and-play que simplifica su configuración y administración.

2.2.3 Estación base

La estación base (*base station*), es un equipo que cuenta con mayores prestaciones, normalmente se trata de un ordenador común, aunque también puede ser un sistema embebido, que se encargará de recoger y almacenar los datos que el gateway previamente recogerá de la red de sensores.

2.3 Aplicaciones de las WSN

Las WSN se utilizan en múltiples escenarios, desde la monitorización de edificios, eficiencia energética, salud personal, control de procesos industriales, ciencia, agricultura o aplicaciones militares.

2.3.1 Entornos militares

Como ya se mencionó con anterioridad, las WSN surgieron dentro del entorno militar y por ello la utilización de esta tecnología en este campo es

puntera. Monitorización de fuerzas y equipos, vigilancia de zonas estratégicas, reconocimiento del terreno, detección de ataques biológicos, químicos o nucleares, etc., son algunos de los ejemplos más empleados.

2.3.2 Procesos agrícolas

El uso de las WSN permite optimizar explotaciones agrarias tanto de invernadero como las de campo abierto. Por ejemplo, se puede controlar remotamente la cantidad de agua, los fertilizantes y los pesticidas que las plantas necesitan, permitiendo a su vez encontrar el momento óptimo de realizar la cosecha, y optimizando la producción, la calidad y diversos aspectos de la cosecha. También pueden diseñarse para alertar de problemas que pueden afectar a las cosechas, tales como heladas, granizo o intrusiones de animales. En la actualidad, en España, existen varias empresas que presentan diversas soluciones al sector de la viticultura.

2.3.3 Estudios medioambientales

Se pueden monitorizar múltiples variables, como temperatura, humedad, actividad sísmica entre otras, permitiendo a científicos y expertos analizar y prevenir problemas o desastres medioambientales. Gracias a estas mediciones los científicos tienen mucha más información para llegar a conocer mejor los contaminantes, los cambios geológicos, las especies invasoras, el flujo del agua, los ciclos de los océanos, la formación de continentes, los lugares que almacenan carbono atmosférico, la razón por la cual los volcanes entran en erupción, etc. Todo ello gracias a miles de sensores distribuidos en grandes áreas permanentes y desatendidas. Un ejemplo sería la Great Duck Island. Great Duck es una isla, ubicada cerca de las costas de Maine, en Estados Unidos. Allí se desplegó una red compuesta por 150 nodos sensores que monitorizan los microclimas en los refugios donde anidan las aves marinas. El propósito es desarrollar un equipo de estudio que permita a los investigadores analizar el hábitat de especies en peligro de extinción sin presencia humana, buscando así no distorsionar la vida de estas aves.

Otra aplicación dentro de este campo es la detección de plagas. Con los sensores adecuados, la red inalámbrica de sensores puede detectar la presencia de plagas en los diferentes entornos a monitorizar. El gran número de sensores, así como su dispersión geográfica, permite la localización de los individuos infestados y su rápido aislamiento para minimizar la dispersión de la misma.

Frente al inquietante problema de degradación del medio ambiente y los recursos naturales ocasionados por los incendios forestales y al alto volumen de pérdidas materiales y humanas el uso de la tecnología WSN puede ayudar a predecir el riesgo de incendio, así como realizar una temprana detección de incendios de forma remota. Para ello, los nodos sensores monitorizarán las variables ambientales que influyen en la creación de incendios, tales como temperatura, humedad, nivel de precipitaciones y velocidad del viento entre otras. Se podrá estimar el riesgo de incendio en las zonas en las que estén desplegados los nodos sensores, permitiendo elaborar planes de contingencia. También resultaría necesario poder analizar a posteriori las causas que han podido provocar el incendio.

2.3.4 Aplicaciones sociales y sanitarias

En pacientes que necesiten tener controladas sus constantes vitales (pulsaciones, presión, nivel de azúcar en sangre, etc.) la monitorización continua en tiempo real, que una red de sensores puede ofrecer, mejoraría notablemente el tiempo de respuesta en la atención urgente de un paciente. Además en situaciones de asistencia en ambulancia los sistemas de telemetría vía cable pueden resultar incómodos e incluso peligrosos. Otra aplicación interesante es la de poder supervisar a los pacientes crónicos estando estos en su propia casa, recogiendo continuamente datos y enviándoselos a su médico, permitiendo un cuidado exhaustivo y así realizar estudios de seguimiento del paciente en profundidad.

Existe un proyecto para asistencia médica de emergencia, que se marca como objetivo la asistencia frente a catástrofes tales como grandes accidentes, fuegos o ataques terroristas. Recibe el nombre de CodeBlue (9).

2.3.5 Procesos industriales

Dentro de este campo las redes de sensores inalámbricas pueden aplicarse en diagnóstico de maquinaria, telemetría, medidas de calidad y monitorizaciones varias, permitiendo así reducir cableado y costes. La reducción del tamaño de estos sensores permite estar allí donde se requiera. Dentro de los procesos de producción industrial se pueden utilizar nodos para "etiquetar" productos, permitiendo conocer en cada momento en que estancia se encuentra cada objeto. También son utilizados para controlar el tiempo entre procesos, para así determinar posibles problemas en la cadena de producción.

2.3.6 Entornos inteligentes

El precio y la facilidad de instalación en edificios y hogares hacen de las WSN una tecnología ideal para su automatización. Se puede controlar la iluminación, la climatización, el riego de jardines, la detección de presencia para economizar energía, por ejemplo, en ascensores que solo estén iluminados cuando haya personas en su interior. Otra aplicación interesante es la seguridad en el hogar a través de la detección de intrusos pudiendo así detectar situaciones de riesgo.

2.3.7 Entornos de alta seguridad

En lugares como centrales químicas, nucleares, aeropuertos o edificios del gobierno que requieren de una seguridad específica para evitar posibles ataques, existen redes con nodos sensores provistos con cámara que envían imágenes a través de la red sin necesidad de conexión directa por cable, con lo que dificultaría un posible sabotaje.

2.3.8 Automoción

Existen diversas soluciones en este campo. Por ejemplo, pueden ser aplicables para realizar comunicaciones entre vehículos, emitiendo señales desde ambulancias, policía o bomberos al resto de vehículos que estén en las inmediaciones, también sería posible comunicarse con el mobiliario urbano destinado a la regulación del tráfico, semáforos, letreros luminosos, etc. También pueden utilizarse para informar a conductores de la situación, en caso de atasco o

accidente, permitiendo así a los usuarios tomar rutas alternativas. También se utilizan en gestión de aparcamientos y comunicación y avisos en cruces entre semáforos y vehículos. Además las redes de sensores pueden ser el complemento ideal a en la vigilancia del tráfico, ya que pueden informar de la situación del tráfico en ángulos muertos que no son cubiertos por las cámaras.

2.3.9 Eficiencia energética

El ahorro energético se ha convertido en una de las cuestiones críticas de nuestros días. Por ello la implantación de diversas medidas e inversiones a nivel tecnológico y de hábitos de consumo en la sociedad cada vez es más común.

La eficiencia energética se basa en la reducción del consumo de energía manteniendo los mismos servicios energéticos, sin disminuir nuestro confort y calidad de vida, protegiendo el medio ambiente, asegurando el abastecimiento y fomentando un comportamiento sostenible en su uso. A día de hoy, los edificios ya consumen aproximadamente el 40% de la energía utilizada en la Unión Europea.

Los dispositivos inalámbricos permiten monitorizar el consumo energético de cualquier sistema/aparato en tiempo real, disgregando información por zona o sistema: aire acondicionado, iluminación, TIC, etc. Las WSN le permiten configurar el sistema en horas y poder reconfigurarlo cuando usted quiera.

El sector de energías renovables presenta unas perspectivas económicas muy interesantes. No obstante, para conseguir la mayor eficiencia es imprescindible una monitorización detallada de los elementos generadores (molinos de viento, paneles solares, etc.). Mediante nodos inalámbricos es posible detectar anomalías en el funcionamiento individual de estos dispositivos, permitiendo su corrección antes del fallo.

Por otro lado, el decreto aprobado por la Directiva 2002/91/CE del Parlamento Europeo y del Consejo, de 16 de diciembre de 2002, abre una puerta muy interesante en la aplicación de las WSN para determinar el grado de eficiencia energética de un edificio o vivienda. Y así, elaborar un certificado para que el

posible comprador del edificio tenga una idea clara sobre el gasto energético del inmueble.

2.4 Características de las WSN

Debido a sus características y su evolución en los últimos años las redes de sensores se han convertido en una tecnología muy interesante. Un mínimo consumo de energía, un bajo coste en su desarrollo y su tecnología inalámbrica son los principales culpables de su éxito. A continuación se muestra en detalle las características más importantes de la tecnología WSN.

2.4.1 Arquitectura distribuida

Por la propia naturaleza de las redes de sensores la arquitectura más apropiada para ellas sigue un paradigma distribuido, de hecho, son redes basadas en la cooperación de los nodos. Los nodos sensores se van a comunicar entre sus nodos vecinos y van a cooperar entre ellos, ejecutando algoritmos distribuidos para obtener una única respuesta global, que un nodo (gateway) se encargará de comunicar a la estación base.

2.4.2 Infraestructura

Las WSN no precisan de una infraestructura de red para poder operar, ya que sus nodos pueden actuar de emisores, receptores o enrutadores de la información. En otras palabras, las redes de sensores son redes *ad-hoc.* Sin embargo, hay que destacar en el concepto de red la figura del nodo recolector (gateway), que es el nodo que recolecta la toda información de la red en un tiempo discreto.

2.4.3 Escalabilidad

Una WSN puede estar compuesta por un número cualquiera de nodos que puede variar desde unos pocos hasta miles de ellos. Además, los nodos pueden moverse en el espacio alterando la topología de la red. Por ello las WSN deben

tener mecanismos robustos que permitan saber en todo momento cómo están distribuidos los nodos en la red y si son accesibles.

2.4.4 Heterogeneidad

Las redes de sensores cuentan con diferentes tipos de dispositivos y es necesario unificarlos dentro de un mismo sistema. La comunicación, su configuración, etc. se deben abstraer de las particularidades del hardware para formar un sistema eficiente de alto nivel.

2.4.5 Sistema en tiempo real

Muchas de las posibles aplicaciones de una red de sensores (prevención de incendios, sistemas de seguridad, etc.) precisan de mecanismos que permitan detectar y actuar en tiempo real a eventos que se produzcan en el entorno. Es decir, un nodo, ante un evento (por ejemplo, alta temperatura debido a la proximidad de un incendio) debe informar de inmediato a la red de este suceso y la red debe hacer llegar el mensaje a la estación base para que se tomen las medidas oportunas.

2.4.6 Tolerancia a fallos

Una de las características más importantes de una red de sensores es la tolerancia a fallos. Un dispositivo sensor dentro de una red tiene que ser capaz de seguir funcionando a pesar de tener errores tanto en el sistema propio como en la red. Si un nodo deja de funcionar el resto de nodos deben ser capaces de auto-configurarse y auto-mantenerse para adaptarse y que la red siga en funcionamiento.

2.4.7 Gestión de recursos y energía

Como ya se ha descrito los sensores son dispositivos limitados en capacidad procesamiento y en energía. Dependiendo de la finalidad de la red desplegada estas limitaciones pueden ser críticas y es por ello que su desarrollo debe realizarse y planificarse con la intención de optimizar sus recursos y maximizar el tiempo de vida de la red de sensores. Se debe conjugar autonomía con capacidad

de proceso: un nodo sensor tiene que contar con un hardware (procesador, radio, sensor, etc.) y un software de consumo ultra bajo. Para poder conseguirlo es necesario que tanto el hardware como el software sean lo más sencillo posible.

2.4.8 Costes de producción reducidos

Dado que la naturaleza de una red de sensores está compuesta de un número elevado de nodos, éstos deben ser económicos. En su idea inicial el costo de cada nodo debería ser lo suficientemente bajo, para que en algunos casos, puedan ser desechados al momento de culminar el estudio o la monitorización que se desea realizar.

2.4.9 Instalación

Otra de las características que hacen de la redes de sensores una tecnología muy atractiva es la facilidad de implantación en cualquier entorno. En lugares cerrados como un edificio no sería necesario realizar ningún tipo de obra para su instalación ya que al tratarse de una tecnología inalámbrica no es preciso realizar ningún tipo de cableado. En lugares abiertos como pueda ser una explotación agraria tampoco necesita ningún tipo de instalación. Simplemente valdría con colocar los nodos para que ellos mismos se configuren para enviar datos por la red.

2.4.10 Seguridad

Al tratarse de una tecnología que utiliza el vacío como canal de comunicación, la información que circula es vulnerable a posibles ataques malintencionados. Además, debido a los escasos recursos de cálculo y almacenamiento de los nodos hacen de la seguridad en las WSN una cuestión crítica ya que los métodos tradicionales de seguridad no se adaptan a estos entornos. Existen ciertos mecanismos de seguridad para las redes de sensores que ofrecen confidencialidad, integridad de los datos y disponibilidad. La seguridad en redes de sensores inalámbricas es un campo que admite una profunda investigación.

2.5 Topologías de red

Existen varias arquitecturas de red que pueden implementarse para aplicaciones WSN como pueden ser estrella, malla o una hibrida entre ellas dos. Cada topología presenta ventajas y desventajas. Para entender cada una de ellas, se detalla a continuación:

- Nodos finales, también llamados RFD (*Reduced Functional Devices*).
- Nodos routers, que dan cobertura a redes de nodos finales y reciben en nombre de FFD (*Full Functional Devices*).
- Nodos gateway, recogen la información de la red y sirven de punto de unión con una LAN o con Internet.

Figura 19: Representación gráfica de los tipos de nodos en una WSN

Topología se refiere a la configuración de los componentes hardware y cómo los datos son transmitidos a través de esa configuración. Cada topología es apropiada bajo ciertas circunstancias y puede ser inapropiada en otras.

2.5.1 Topología en estrella

Una topología en estrella es un sistema donde la información enviada sólo da un salto y donde todos los nodos sensores están en comunicación directa con la puerta de enlace (gateway), usualmente a una distancia de 30 a 100 metros. Todos los nodos sensores son idénticos, nodos finales, y el gateway capta toda la información de los nodos. Los nodos finales no intercambian información entre ellos.

En términos energéticos la topología en estrella es la que tiene un menor gasto, pero se ve muy limitada por la distancia de transmisión entre cada nodo y el gateway. Además, otro problema es que los nodos no cuentan con un camino

alternativo (multipath) para poder comunicarse con el gateway por lo que podría perderse información del nodo sensor si el enlace con el gateway fallase.

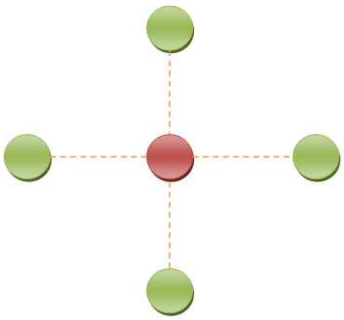


Figura 20: Ejemplo de WSN con topología en estrella

2.5.2 Topología en malla

La topología malla es un sistema multisalto, donde todos los nodos son routers. Cada nodo puede enviar y recibir información de otro nodo y del gateway, a diferencia de la topología en estrella. La propagación de los datos a través de los nodos hacia la puerta de enlace hace posible, por lo menos en teoría, crear una red con una extensión ilimitada. También es altamente tolerante a fallos ya que cada nodo tiene diferentes caminos para comunicarse con la puerta de enlace. Si un nodo falla, la red se reconfigurará alrededor del nodo fallido automáticamente. Dependiendo del número de nodos y de la distancia entre ellos, la red puede experimentar periodos de espera elevados a la hora de mandar la información.

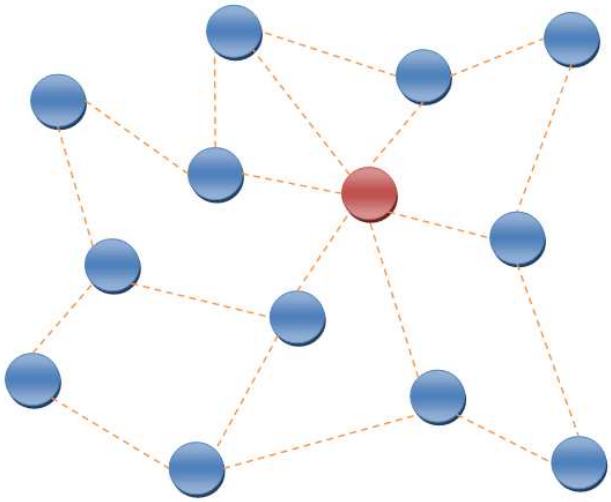


Figura 21: Ejemplo de WSN con topología en malla

2.5.3 Topología hibrida estrella-malla

Este tipo de red busca combinar las ventajas de los otros dos tipos, la simplicidad y el bajo consumo de una topología en estrella, así como la posibilidad de cubrir una gran extensión y de reorganizarse ante fallos de la topología en malla. Crea una red en estrella alrededor de los nodos routers pertenecientes a una red en malla. Los routers dan la posibilidad de ampliar la red y de corregir fallos en estos nodos y los nodos finales se conectan con los routers cercanos ahorrando energía.

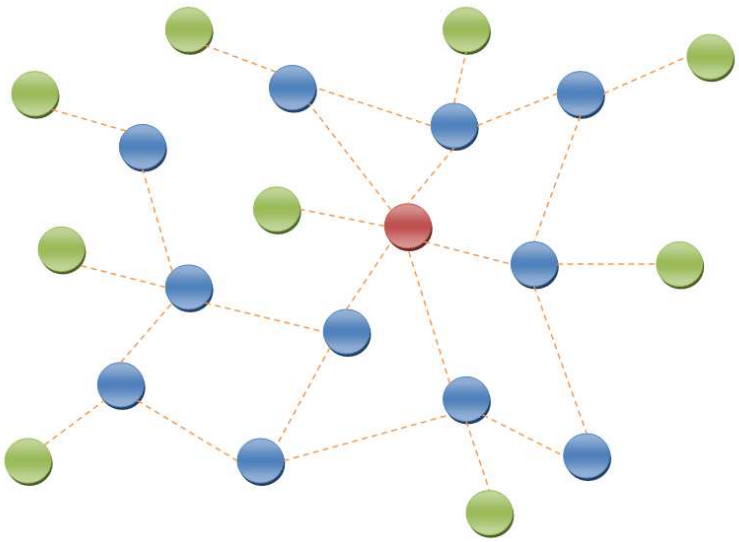


Figura 22: Ejemplo de WSN con topología híbrida estrella-malla

2.5.4 Topología en árbol

En la topología en árbol, cada nodo se conecta a un nodo de mayor jerarquía en el árbol y así sucesivamente hasta llegar al nodo gateway. Es decir, los datos son enrutados desde el emisor a nodos de mayor jerarquía en el árbol hasta alcanzar el gateway. Esta topología ofrece una mayor eficiencia en la recolección de datos de la red debido a que los datos recorren el camino más eficiente hasta llegar al nodo gateway. Por otro lado, no dispone de caminos alternativos y en caso de que un nodo de la red fallase todos los nodos que se encuentren por debajo en la jerarquía se quedarán aislados.

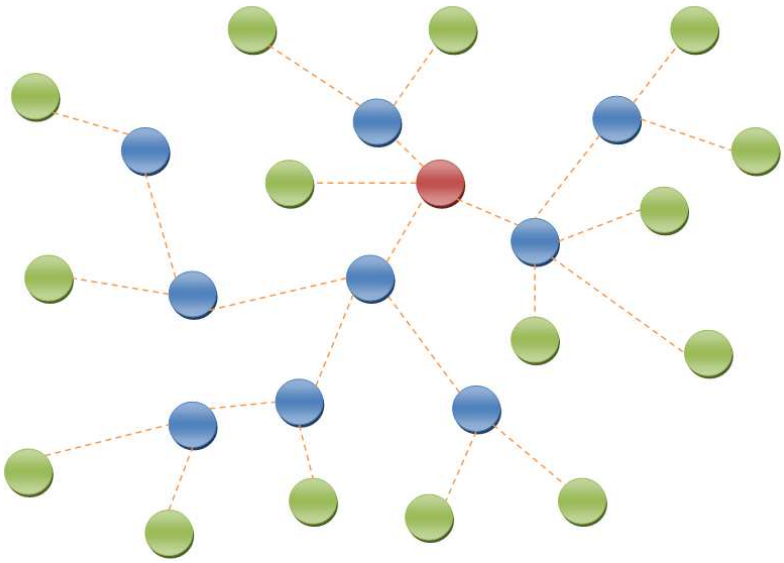


Figura 23: Ejemplo de WSN con topología árbol

2.6 Estándar de comunicación ZigBee

Los protocolos de comunicación existentes (WiFi, Bluetooth, etc.) y las características particulares de los nodos sensores no permitían una integración satisfactoria de ambas tecnologías. Por ello, surge un consorcio de empresas (Honeywell, Invensys, Mitsubishi, Motorola, Philips, Samsung, etc.) con el objetivo de definir las especificaciones de un nuevo protocolo de comunicación inalámbrica, basado en la norma IEEE 802.15.4, que se adapte a las necesidades y limitaciones de los motes (fiable, económico y de mínimo consumo).

El consorcio recibiría el nombre de ZigBee Alliance presentando a finales de 2004 la primera especificación del estándar IEEE 802.15.4 que define los protocolos de alto nivel de comunicación para su utilización con radios digitales para redes de área personal inalámbricas (WPAN).

La Tabla 1 muestra una comparativa de los distintos estándares de comunicación existentes en el mercado. Ilustra detalles como la topología, la frecuencia de radio empleada, rango, etc.

Tabla 1: Comparativa de estándares de comunicación inalámbrica

Wireless Networking Technologies						
	ZigBee	Bluetooth	UWB	Wi-Fi	LonWorks	Proprietary
Standard	IEEE 802.15.4	IEEE 802.15.1	IEEE 802.15.3a (to be ratified)	IEE 802.11a, b, g (n to be ratified)	EIA 709.1, 2, 3	Proprietary
Industry organizations	ZigBee Alliance	Bluetooth SIG	UWB Forum and WiMedia Alliance	Wi-Fi Alliance	LonMark Interoperability Association	N/A
Topology	Mesh, star, tree	Star	Star	Star	Medium-dependent	P2P, star, mesh
RF frequency	868/915 MHz, 2.4 GHz	2.4 GHz	3.1 to 10.6 GHz (U.S.)	2.4 GHz, 5.8 GHz	N/A (wired technology)	433/868/900 MHz, 2/4 GHz
Data rate	250 kbits/s	723 kbits/s	110 Mbits/s to 1.6 Gbits/s	11 to 105 Mbits/s	15 kbits/s to 10 Mbits/s	10 to 250 kbits/s
Range	10 to 300 m	10 m	4 to 20 m	10 to 100 m	Medium-dependent	10 to 70 m
Power	Very low	Low	Low	High	Wired	Very low to low
Battery operation (life)	Alkaline (months to years)	Rechargeable (days to weeks)	Rechargeable (hours to days)	Rechargeable (hours)	N/A	Alkaline (months to years)
Nodes	65,000	8	128	32	32,000	100 to 1000

Se espera que los módulos ZigBee sean los transmisores inalámbricos más baratos jamás producidos de forma masiva. Disponen de una antena integrada, control de frecuencia y una pequeña batería. Además de ser el estándar aceptado y utilizado por las WSN, ZigBee es un sistema ideal en el uso de redes domóticas y especialmente diseñado para reemplazar la proliferación de sensores y actuadores individuales.

El estándar IEEE 802.15.4 define la capa física (PHY) y la de acceso al medio (MAC) para estos dispositivos. Ofrece soporte para topología en estrella y multisalto cuando los enlaces superan los 10 metros.

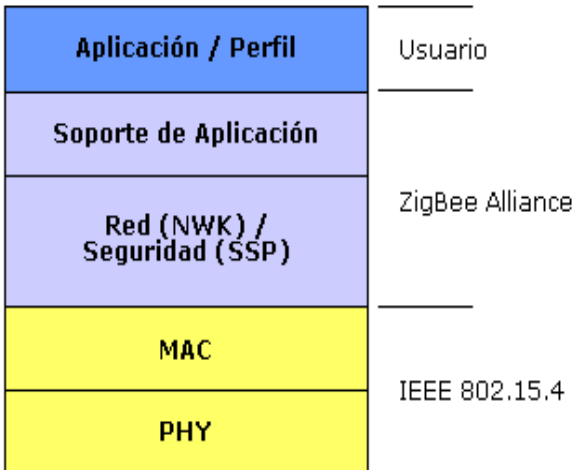


Figura 24: Pila de protocolo del estándar ZigBee

2.6.1 Capa física (PHY)

Dentro de la capa física se definen dos opciones, las dos basadas en DSSS (*Direct Sequence Spread Spectrum*). La banda global de 2.4 GHz y la de 868/915 MHz, disponibles en Europa y Norte América respectivamente. La banda de 2.4 GHz tiene 16 canales con una tasa de 250 Kbps, la de 915 MHz tiene 10 con una tasa de 40 Kbps y la de 868 MHz un canal de 20 Kbps. La siguiente tabla muestra un resumen de las bandas de radio:

Tabla 2: Resumen de las bandas de radio empleadas en el estándar ZigBee

PHY (MHz)	Banda de Frecuencia (MHz)	Tasa de Chip (Chips/s)	Modulación	Tasa de Bit (kbps)	Tasa de Símbolo (ksímbolos/s)
868	868-868.6	300	BPSK	20	20
915	902 - 928	600	BPSK	40	40
2450	2400 - 2483.5	2000	O-QPSK	250	62.5

2.6.2 Capa de control de acceso al medio (MAC)

La capa MAC controla todo el acceso al canal de radio. Ofrece una interfaz entre la capa de aplicación y la capa PHY. La capa MAC provee servicios a la capa de aplicación a través de dos grupos:

- MAC Data Service (MCPS): provee servicios de transporte de datos entre pares de MACs.

- Management Service (MLME): provee interfaces a través de las cuales las funciones de la capa de gestión pueden ser invocadas. También accede a los servicios de MCPS para el transporte de datos.

La capa MAC se encarga de las siguientes tareas:

- Generación de balizas de red si el dispositivo es un coordinador.
- Sincronización de las balizas.
- Apoyo a la asociación y la disociación del PAN.
- Dispositivo de apoyo a la seguridad.
- Que utilicen el mecanismo CSMA-CA de acceso al canal.
- Manejo y mantenimiento del mecanismo de GTS.
- Proporcionar un vínculo fiable entre dos entidades pares MAC.

Otra de las características del control de acceso al medio para el estándar 802.15.4 es que proporciona 26 primitivas, mientras que Bluetooth tiene alrededor de 131 primitivas en 32 eventos. Se trata de un estándar ligero pensado para las estrictas necesidades de la comunicación entre nodos sensores.

2.7 Sistemas operativos

Los nodos sensores necesitan de un sistema operativo (SO) para que actué de interfaz entre el hardware y las aplicaciones de usuario. El sistema operativo es responsable de gestionar, y coordinar las actividades y llevar a cabo el intercambio de los recursos del nodo sensor. Se dispone de diversos sistemas operativos de los cuales aquí se enumeran los más importantes:

- **TinyOS** (10): sistema operativo desarrollado por la universidad de Berkeley. Es considerado el estándar *de facto* en sistemas operativos para nodos sensores. El siguiente capítulo describe extensamente este sistema operativo.
- **Contiki** (11): es un sistema operativo de libre distribución para usar en un limitado tipo de computadoras, desde los 8 bits a sistemas embebidos en microcontroladores, incluidas motas de redes inalámbricas.

- **MANTIS (MultimodAl NeTworks In-situ Sensors)** (12): sistema operativo para nodos sensores desarrollado por la universidad de Colorado.
- **LiteOS** (13): sistema operativo desarrollado en principio para calculadoras, pero que ha sido también utilizado para redes de sensores.
- **Bertha (pushpin computing platform)** (14): una plataforma de software diseñada e implementada para modelar, testear y desplegar una red de sensores distribuida de muchos nodos idénticos.
- **CORMOS** (15): Communication Oriented Runtime System for Sensor Networks, específico para redes de sensores inalámbricos como su nombre indica.
- **eCos** (16) (embedded Configurable operating system): es un sistema operativo gratuito, en tiempo real, diseñado para aplicaciones y sistemas embebidos que sólo necesitan un proceso. Se pueden configurar muchas opciones y puede ser personalizado para cumplir cualquier requisito, ofreciendo la mejor ejecución en tiempo real y minimizando las necesidades de hardware.
- **MagnetOS** (17): es un sistema operativo distribuido para redes de sensores o ad-hoc, cuyo objetivo es ejecutar aplicaciones de red que requieran bajo consumo de energía, adaptativas y fáciles de implementar.
- **Nut/OS** (18): es un pequeño sistema operativo para aplicaciones en tiempo real, que trabaja con CPUs de 8 bits.
- **t-Kernel** (19): es un sistema operativo que acepta las aplicaciones como imágenes de ejecutables en instrucciones básicas. Por ello, no importará si está escrito en C++ o lenguaje ensamblador.

2.8 Lenguajes de programación

La programación de sensores es relativamente compleja. Entre sus dificultades está la limitada capacidad de cómputo y escasa la cantidad de recursos. Y así como en los sistemas informáticos tradicionales se encuentran entornos de programación prácticos y eficientes para generar y depurar código,

incluso en tiempo de ejecución, la programación para estos microcontroladores todavía no hay herramientas comparables. Podemos encontrar lenguajes como:

- **C@t**: iniciales que hincan computación en un punto del espacio en el tiempo (Computation at a point in space (@) Time)
- **DCL**: lenguaje de composición distribuido (Distributed Compositional Language)
- **GalsC**: diseñado para ser usado en TinyGALS, es un lenguaje programado mediante el modelo orientado a tarea, fácil de depurar, permite concurrencia y es compatible con los módulos nesC de TinyOS
- **nesC** (20): lenguaje que utilizamos para nuestras motas, y que está directamente relacionado con TinyOS.
- **Protothreads**: aunque no se trata de un lenguaje en sí, provee los mecanismos necesarios para la programación concurrente en el sistema operativo Contiki, provee hilos de dos bytes como base de funcionamiento.
- **SNACK**: facilita el diseño de componentes para redes de sensores inalámbricas, sobre todo cuando la información o cálculo a manejar es muy voluminoso, complicado con nesC, este lenguaje hace su programación más eficiente. Luego es un buen sustituto de nesC para crear librerías de alto nivel a combinar con las aplicaciones más eficientes.
- **SQTL** (Sensor Query and Tasking Language): como su nombre indica es una interesante herramienta para realizar consultas sobre redes de sensores.

Capitulo 3 Entorno de desarrollo

En este capítulo se detalla el entorno de desarrollo utilizado para la realización de este proyecto. Se dividirá en dos partes, hardware y software.

3.1 Hardware

Los componentes hardware que se han usado en este proyecto son el nodo basado en la plataforma MicaZ, la placa sensora MTS310CB y la placa programadora MIB520 todos ellos fabricados por Crossbow Technology Inc.

3.1.1 Nodo MicaZ

Los nodos o motes MicaZ (21) (módulo MPR2400CA) son una plataforma hardware diseñada para el desarrollo de múltiples de aplicaciones para redes de sensores inalámbricas (véase apartado 2.3 Aplicaciones de las WSN).

Figura 25: Nodo MicaZ de Crossbow Technology Inc

En el apartado técnico, cada mote MicaZ incorpora un microprocesador ATMEL Mega128L de 7 MHz. Dicho procesador incorpora una pequeña memoria EEPROM de 128 KB dedicado a programas, una memoria RAM de 4 KB para datos. Además, dispone de una memoria flash externa de 512 KB para aquellas aplicaciones que necesiten guardar datos de manera permanente.

El microprocesador se conecta, a través de un Bus SPI, con la radio CC2420, fabricada por Texas Instruments, con una tasa transferencia de datos de hasta 250

Kbps. Todo ello a través de la interfaz 802.15.4 compatible con ZigBee. Según la documentación del fabricante la radio cuenta con una cobertura de 75 a 100 metros en espacios abiertos mientras que para interiores es de 20 a 30 metros. Los consumos de la radio son de 19,7mA en modo activo, 20μA modo inactivo y 1μA en modo hibernación.

Por otro lado, cada MicaZ posee un conector de expansión de 51 pins. En él se pueden conectar las placas de sensores, las placas programadoras y otros dispositivos de entrada/salida. Además incorpora una interfaz formada por tres diodos LEDs (rojo, amarillo y verde) que actúa como ventana sobre lo que ocurre durante el funcionamiento de un nodo. Las dimensiones del nodo MicaZ son 58x32x7mm y pesa 18gr excluyendo las baterías (dos pilas AA). La batería de los motes deben proporcionan un voltaje entre 2.7 y 3.3 voltios. Según la documentación del fabricante (22) los consumos del microprocesador son de 8mA en modo activo, mientras que en modo hibernación es menor a 15μA.

3.1.2 Placa Sensora MTS310CB

La MTS310CB (23) es una placa sensora flexible con una gran variedad de modalidades de detección. Estas modalidades incluyen un acelerómetro de dos ejes (Honeywell HMC1002), un magnetómetro de doble eje (ADXL202JE), luz, temperatura (Panasonic ERT-J1VR103J), zumbador y micrófono. Esta placa puede ser usada en plataformas Iris, MicaZ y Mica2.

Figura 26: Placa Sensora MTS310CB de Crossbow Technology Inc

Tiene un tamaño de 56x11x31mm y una de las características de esta placa sensora es que comparte el mismo canal conversor Analógico/Digital (ADC1) por lo que solo se puede activar un sensor a la vez, de lo contrario, la lectura en dicho canal será corrupta y no tendrá ningún sentido.

Con el objetivo de ayudar a minimizar el consumo de energía, estos sensores vienen equipados con mecanismos de control de la energía. Por defecto, el sensor estará apagado.

3.1.3 Placa programadora MIB520CB

La placa programadora MIB520CB (24) tiene dos funcionalidades básicas. Por un lado permite cargar el código ejecutable en los motes, y por otro, puede funcionar como puerta de enlace entre los motes y el PC (gateway).

Figura 27: Placa programadora MIB520CB de Crossbow Technology Inc

El MIB520CB, al igual que la placa sensora MTS310CB, puede ser usada en plataformas Iris, MicaZ y Mica2. Esta placa programadora ofrece conectividad USB

para la comunicación entre nodos y la estación base así como la programación de los nodos. Para funcionar como gateway la placa programadora necesita acoplarse con un nodo sensor, previamente programado, a través del conector de 51 pins. La placa incorpora tres LEDs (amarillo, verde y rojo) que nos indica si la placa está conectada a la corriente eléctrica (verde) y si se están cargando los datos en el mote acoplado desde el PC (rojo). También incorpora un interruptor para deshabilitar la transmisión en serie con los motes, y un botón para reiniciar la placa programadora. La placa se alimenta con 5 voltios y unos 50mA de corriente a través del cable USB conectado al PC.

3.2 Software

En esta sección se describen los distintos elementos software (sistemas operativos, base de datos, lenguajes de programación, etc.) que intervienen en la construcción de la aplicación para redes de sensores inalámbricas desarrollada en este proyecto.

3.2.1 NesC

nesC (Network Embedded Systems C) es un lenguaje de programación basado en C, enfocado y optimizado para su uso en aplicaciones de redes de sensores. Su origen se debe al deseo de disponer de un lenguaje específico que cumpliese con el modelo de ejecución y los conceptos del sistema operativo TinyOS, el cual se describe a continuación. No fue hasta la versión 1.0 de TinyOS cuando se reescribió todo el código en nesC. En versiones anteriores (por ejemplo, la versión 0.6) de TinyOS todo el sistema operativo y la programación de aplicaciones fue desarrollada en C.

Esta necesidad de desarrollar un nuevo lenguaje de programación específico para redes de sensores inalámbricos viene motivada por el tipo de aplicaciones que se desarrolla, que se caracterizan por:

- Son aplicaciones basadas en recolección, difusión y control de la información obtenida del sensor, es decir, no son aplicaciones de propósito general.

- Tienen que reaccionar ante cambios en su entorno (eventos).
- Es preciso optimizar la limitada cantidad de recursos que ofrecen los nodos (memoria, capacidad de computo, consumo de energía)
- Deben ser aplicaciones estables, puesto que deben correr durante meses/años sin intervención humana.
- Precisan de control de errores en la manejo de datos.
- Son aplicaciones en tiempo real (envío de mensajes a la red).

Teniendo en cuenta estos aspectos, nesC aporta como concepto innovador la metodología de lenguaje orientado a componentes. Con respecto al lenguaje C tradicional, en nesC, se precisan restricciones teniendo en cuenta a las limitaciones de los nodos. Estas precondiciones estáticas del lenguaje, son la base de la optimización y permiten al compilador realizar análisis profundos sobre el código. Además nesC permite desarrollar *interfaces* y *componentes* y estos últimos pueden ser a su vez módulos o configuraciones. A continuación se describen los constructores del lenguaje.

3.2.1.1 Interfaces

Cada componente suele implementar una serie de servicios que son ofrecidos al exterior mediante una interfaz. Las interfaces son bidireccionales, ofreciendo *comandos* y *eventos*, siendo básicamente en ambos casos funciones que son invocadas por la aplicación o por el hardware u otros componentes de nivel respectivamente.

Los comandos son llamadas a funciones del componente, tal y como puede ser “encender o parar un temporizador”. Por otro lado, los eventos son notificaciones del componente, que indican que una tarea ha sido completada, tal y como puede ser la “confirmación del envío de un mensaje”. A continuación se muestra un ejemplo de fichero interfaz:

```
interface Timer {
      command result_t start(char type, uint32_t interval);
      command result_t stop();
      event result_t fired();
}
interface SendMsg {
      command result_t send(TOS_Msg *msg, uint16_t length);
      event result_t sendDone(TOS_Msg *msg, result_t success);
}
```

3.2.1.2 Componentes

Una aplicación en nesC está formada por varios componentes. Un componente implementa una serie de servicios que ofrece al exterior mediante una interfaz. Además, un componente puede usar las interfaces de otros componentes para su propio uso. Existen dos tipos de componentes:

- Módulo.
- Configuración.

3.2.1.2.1 Módulo

Los módulos contienen el código que implementa los servicios que ofrece el componente. En el siguiente ejemplo se puede apreciar la declaración y la implementación de los servicios. A continuación se muestra un ejemplo de un módulo:

```
module TimerM {
      provides {
            interface StdControl;
            interface Timer;
      }
      uses interface Clock;
      }
      implementation {
            command result_t Timer.start(char type, uint32_t
            interval){
                  ...
            }
      command result_t Timer.stop() {
      ...
      }
      event void Clock.tick() {
      ...
      }
}
```

3.2.1.2.2 Configuración

La configuración permite cablear (*wire*) o asociar diferentes componentes mediante interfaces para que trabajen en conjunto como si fueran uno solo (supercomponente), es decir, un componente de mayor abstracción. Para crear un programa en nesC, es preciso seleccionar los componentes y "linkarlos" a las interfaces que estos componentes proveen. Esta operación, se denomina *wiring*. A continuación se muestra un ejemplo del fichero de configuración:

```
configuration TimerC {
      provides {
            interface StdControl;
            interface Timer;
      }
}
implementation {
      components TimerM, HWClock;
      // Pass-through: Connect our "provides" to TimerM "provides"
      StdControl = TimerM.StdControl;
      Timer = TimerM.Timer;
      // Normal wiring: Connect "requires" to "provides"
      TimerM.Clock -> HWClock.Clock;
}
```

El siguiente código muestra un ejemplo de cómo se realiza un *wiring*:

```
configuration EjemploAppC
{
}
implementation
{
  components MainC, EjemploC;
  EjemploC.Boot -> MainC;
}
```

En el fichero de configuración se indican los componentes a utilizar, uno de ellos siempre debe ser el componente al que dicha configuración implementa, que en este caso es `EjemploC`. Por otro lado, `MainC` es el componente que pretende ser utilizado para esta aplicación. Por lo tanto debe "linkarse", y para ello es precisa la utilización de una interfaz, que en este caso es `Boot`. Es decir, que el componente `EjemploC` utilizará el componente `MainC` a través de la interfaz `Boot`. Para indicar este enlace se realiza con el operador "->".

Es necesario indicar en el fichero del módulo las interfaces que el componente, a implementar, va a utilizar para acceder a otros componentes, que

en este caso es `Boot`. Por último, para utilizar un componente a través de un interfaz, dicho componente debe proveer dicha interfaz, siguiendo con el ejemplo:

```
module EjemploC(){
  uses interface Boot;
}
implementation{
      …
}
```

3.2.1.3 Modelo de concurrencia

Puesto que las redes de sensores inalámbricos poseen eventos asíncronos, que deben tener prioridad sobre otras tareas, se hace necesario el uso de un modelo de concurrencia que permita detener la ejecución de parte de un código menos prioritario, para atender un evento asíncrono de mayor relevancia. Si no se tiene en cuenta dichos eventos asíncronos se pueden producir las denominadas "condiciones de carrera" (o *data races*), en la que varios procesos intentan utilizar información compartida, haciendo que esta se pierda o quede corrupta por modificaciones descontroladas sobre los datos.

Las tareas (*tasks*) en nesC son la solución a los problemas de compartición de información. Son funciones atómicas, que se ejecutan sin permitirse las interrupciones entre tareas, aunque sí por eventos. Cuando aparece un evento asíncrono se analiza y se pospone la ejecución de su código atómico, para seguir permitiendo la recepción de otros eventos.

3.2.1.3.1 Condiciones de carrera

Como en cualquier lenguaje de programación orientado a sistemas de tiempo real, nesC tiene mecanismos para detectar y prevenir las condiciones de carrera, se clasifican en código fuente en dos tipos:

- Código asíncrono: código al que se llega por una interrupción.
- Código síncrono: código al que se llega desde las tareas.

Si en tiempo de ejecución se intenta acceder a una variable compartida a causa de un evento asíncrono, se produce una posible condición de carrera que se debe analizar. Sin embargo, el código síncrono es atómico con respecto al código

síncrono, es decir, que la ejecución de código síncrono, nunca se verá interrumpida por otro código síncrono y no se correrá el riesgo de condiciones de carrera.

En resumen, nos encontraremos con posibles condiciones de carrera únicamente ante la ejecución de código asíncrono. El compilador de nesC tiene constancia de este hecho, y gracias a la traza de las interrupciones es capaz de detectar las condiciones de carrera.

```
// Signaled by interrupt handler
event void Receive.receiveMsg(TOS_Msg *msg) {
      if (recv_task_busy) {
            return; // Drop!
      }
      recv_task_busy = TRUE;
      curmsg = msg;
      post recv_task();
}
task void recv_task() {
      // Process curmsg ...
      recv_task_busy = FALSE;
}
```

Para solucionar una situación de condiciones de carrera, se pueden plantear varias alternativas:

- Trasladar el acceso a las variables compartidas a tareas (*tasks*).
- Usar secciones de código *atomic*, que indican al compilador que esa sección no puede ser interrumpida por un evento asíncrono.

Para alcanzar estas soluciones se pueden usar varios métodos que van desde la inhibición de interrupciones hasta el uso de semáforos, siendo la primera de ellas la que se usa en la actualidad. Hay que tener cuidado con el uso de estas técnicas ya que puede generar que se pierda información al no permitir la entrada de eventos asíncronos durante un largo tiempo.

3.2.1.4 Programando en nesC

Existen una serie de sugerencias de programación para evitar los problemas comunes que surgen a la hora de programar aplicaciones de redes de sensores en nesC. Estas sugerencias han sido extraídas de TinyOS Programming (25). A continuación se enumeran dichos consejos:

- **Sugerencia 1**: debe evitarse hacer llamadas a eventos desde la implementación de un comando, ya que podría provocar un bucle de llamada muy largo, podría corromper la memoria y provocar la caída del programa.
- **Sugerencia 2**: las tareas deben ser cortas.
- **Sugerencia 3**: programar código sincrónico siempre que sea posible. El código debe ser asincrónico sólo si el tiempo es un recurso limitado.
- **Sugerencia 4**: se debe utilizar pocas secciones atómicas y estas deben ser cortas. Hay que tener cuidado en llamar a otros componentes dentro de una sección atómica.
- **Sugerencia 5**: sólo uno de los componentes de una aplicación debe ser capaz de modificar los datos de un puntero. En el mejor de los casos, sólo uno de los componentes deben almacenar el puntero en cualquier momento.
- **Sugerencia 6**: si los requisitos de la aplicación requieren una pila o una cola de memoria dinámica, sería conveniente encapsularlo en un componente y tratar de limitar sus accesos.
- **Sugerencia 7**: conservar la memoria mediante el uso de las enumeraciones en lugar de constantes enteras, y no declarar las variables como un tipo de enumeración.
- **Sugerencia 8**: en la parte superior de configuración de nivel de abstracción de software, auto-cableado `Init` para `MainC`. Esto elimina la carga de la inicialización de cableado para el programador, que elimina el trabajo innecesario de la secuencia de arranque y elimina la posibilidad de errores de olvidarse del cableado.
- **Sugerencia 9**: si un componente es una abstracción utilizable como tal, su nombre debe terminar con C. Si se pretende ser una parte interna y privada de una mayor abstracción, su nombre debe terminar con P. Nunca debe cablearse los componentes de P desde el exterior su paquete (directorio).
- **Sugerencia 10**: usar la palabra clave "as" para especificar alias de componentes o interfaces.

- **Sugerencia 11**: en tiempo de compilación nunca ignore las advertencias.
- **Sugerencia 12**: si una función tiene un argumento que es un pequeño número de constantes, considere la posibilidad de definir como un par de funciones por separado para evitar errores. Si las funciones de una interfaz, todos tienen un argumento que es casi siempre una constante dentro de una amplia gama, considere el uso de una interfaz de parámetros para ahorrar espacio de código. Si las funciones de una interfaz, todos tienen un argumento que es una constante dentro de un rango grande, pero sólo ciertos valores válidos, su aplicación como una interfaz de parámetros, pero exponerlo como interfaces individuales, tanto para minimizar el tamaño del código y evitar errores.
- **Sugerencia 13**: nunca utilizar el atributo "`packet`".
- **Sugerencia 14**: utilizar siempre los tipos de datos de plataforma independiente (`nx_`) en la definición de formatos de mensaje.
- **Sugerencia 15**: si tiene que realizar el cálculo significativo en un tipo independiente de la plataforma o muchos accesos (cientos o más) veces, copiar temporalmente a un tipo nativo puede ser una buena idea.

3.2.2 Java

El objetivo de este apartado no es dar una visión completa y profunda sobre el lenguaje de programación Java, ya que esta fuera del alcance del presente proyecto. Se trata pues, de realizar un breve resumen de referencia debido a que durante la realización de este proyecto se vio como la herramienta Xserve (26) que viene incluida en el kit de desarrollo de Crossbow Technology Inc (27) no es compatible con la versión de TinyOS 2.x. Por lo tanto, se desarrolló una aplicación Java que realizase las mismas funciones que Xserve.

Java es un lenguaje de programación desarrollado por Sun Microsystems (28). Es un lenguaje orientado a objetos y toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel. Las aplicaciones Java están típicamente compiladas en un *bytecode* y en tiempo de

ejecución, el *bytecode* es interpretado o compilado a código nativo para su ejecución.

Desde noviembre de 2006 Sun Microsystems liberó la mayor parte de sus tecnologías Java se encuentran bajo la licencia GNU GPL (29) de forma que prácticamente todo el Java de Sun es ahora software libre (aunque la biblioteca de clases de Sun que se requiere para ejecutar los programas Java aún no lo es).

La elección de usar Java para la implementación de la aplicación que sustituye a Xserve de Crossbow Technology Inc se debe a que TinyOS se encuentra estrechamente relacionado con dicho lenguaje de programación. La mayoría de herramientas de apoyo de las aplicaciones de TinyOS están realizadas en Java.

3.2.3 TinyOS 2.x

TinyOS es un sistema operativo de código abierto (*open source*) diseñado para responder a las características y necesidades de las redes de sensores, tales como las restricciones de memoria, bajo consumo de energía, operaciones de concurrencia intensiva, diversidad en diseños y usos. Además se encuentra optimizado en términos de uso de memoria y eficiencia de energía. Su arquitectura está pensada para incorporar nuevas funcionalidades de manera rápida y sencilla. Como ya se ha mencionado está escrito en nesC. TinyOS es en definitiva una biblioteca de componentes en el que se incluyen multitud de servicios (control de acceso a medio, protocolos de red, drivers de sensores, herramientas de manejo de datos, etc*.)* que pueden ser utilizados tal cual o pueden ser modificados para aplicaciones que lo precisen.

TinyOS se ha convertido, por sus características, en el estándar de facto en sistemas operativos para nodos sensores. Fue desarrollado a partir de la Tesis Doctoral de Jason Lester Hill (30) en 2003 por la Universidad de Berkeley en California. Desde entonces ha crecido hasta tener una comunidad extensa e internacional de desarrolladores y usuarios.

3.2.3.1 Características generales

TinyOS es un sistema operativo compuesto por un pequeño núcleo multitarea, útil para dispositivos embebidos, como los motes. Su diseño está compuesto por los siguientes niveles de abstracción:

- **Eventos**: diseñados para realizar procesos pequeños. Por ejemplo cuando el contador del *timer* se interrumpe, o atender las interrupciones de un conversor análogo-digital. Los eventos tiene mayor prioridad y pueden interrumpir cualquier tarea que se esté ejecutando.
- **Tareas**: las tareas están diseñadas para realizar una cantidad mayor de procesamiento y, a diferencia de los eventos, no son críticas en tiempo. Por ejemplo calcular el promedio de varias mediciones tomadas de un sensor. Las tareas se ejecutan en su totalidad, pero la solicitud de iniciar una tarea, y el término de ella son funciones separadas. Esta característica es propia de la programación orientada a componentes.
- **Comandos**: son llamadas a la implementación de los comandos de los componentes, es decir, un componente llama a comandos que pertenecen a otros componentes que se encuentran en capas inferiores.

Con este diseño se permite que los eventos puedan ser ejecutados inmediatamente, pudiendo interrumpir a las tareas. El enfoque basado en eventos es la solución ideal para alcanzar un alto rendimiento en aplicaciones de concurrencia intensiva. Adicionalmente, este enfoque usa las capacidades de la CPU de manera eficiente y de esta forma gasta el mínimo de energía.

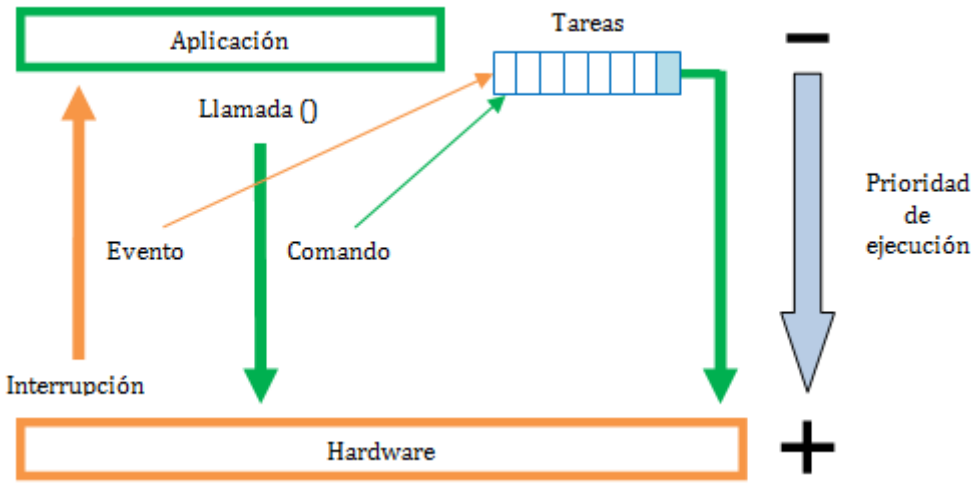


Figura 28: Esquema funcional de TinyOS

Como puede verse en la Figura 28 las dos fuentes de concurrencia en TinyOS son las tareas y los eventos. Los componentes entregan tareas al planificador, siendo el retorno de éste inmediato, aplazando el cómputo hasta que el planificador ejecute la tarea. Los componentes pueden realizar tareas siempre y cuando los requerimientos de tiempo no sean críticos. Para asegurar que el tiempo de espera no sea muy largo, se recomienda programar tareas cortas, y en caso de necesitar procesamientos mayores, se recomienda dividirlo en múltiples tareas. Las tareas se ejecutan en su totalidad, y no tiene prioridad sobre otras tareas. Así también los eventos hasta completarse, pero estos sí pueden interrumpir otros eventos o tareas, con el objetivo de cumplir de la mejor forma los requerimientos de tiempo real.

Todas las operaciones de larga duración deben ser divididas en dos estados: la solicitud de la operación y la ejecución de ésta. Esta operación se denomina *split-phase*. Si un comando solicita la ejecución de una operación, éste debe retornar inmediatamente mientras que la ejecución queda en manos del planificador, el cual deberá señalizar a través de un evento, el éxito o error de la operación.

Una ventaja secundaria de elegir el modelo basado en componentes, es que propaga las abstracciones del hardware en el software. Tal como el hardware responde a cambios de estado en sus pines de entrada/salida, los componentes responden a eventos y a los comandos en sus interfaces.

3.2.3.2 Evolución y cambios de diseño en TinyOS 2.x

TinyOS 2.x es un rediseño completo de la primera versión. Su desarrollo fue motivado porque muchos aspectos de la versión 1.x se esfuerzan por satisfacer las necesidades que, en un principio, no fueron previstos cuando fue diseñado e implementado. La estructura e interfaces definidas en 1.x tienen varias limitaciones fundamentales, si bien con estas limitaciones se puede trabajar de todos modos, la práctica de subsanar las deficiencias de TinyOS 1.x ha conducido que tuviese componentes fuertemente acoplados, encontrando interacciones entre ellos. La versión 1.x de TinyOS supone un duro aprendizaje para un recién llegado a la programación de redes de sensores.

La versión 2.x no es compatible con 1.x, es decir, el código escrito para 2.x no es posible compilarlo para 1.x., siendo la portabilidad de aplicaciones un activo ámbito de investigación. Sin embargo, se ha tratado de minimizar la dificultad que requiere a la hora de migrar el código de una aplicación de 1.x a 2.x. Su última versión (TinyOS 2.1.0, utilizada en este proyecto) fue liberada en Agosto de 2008.

3.2.3.2.1 Abstracciones de hardware

Una de la mas importantes mejoras de TinyOS 2.x con respecto a la versión predecesora es la organización de las abstracciones de hardware, llamado "Hardware Abstraction Architecture" (HAA), en TinyOS 2.x generalmente siguen una jerarquía de tres niveles de abstracción (ver Figura 29).

La capa inferior de la HAA es "Hardware Presentation Layer" (HPL). El HPL es la capa inferior de software en la parte superior del hardware, presentando el hardware como pins I/O o los registros como interfaces nesC. El HPL generalmente no tiene estado ni tiene variables. Los componentes HPL suelen tener el prefijo Hpl, seguido por el nombre del chip. Por ejemplo, el componente de HPL de la CC2420 es `HplCC2420`.

En la parte intermedia de HAA está "Hardware Adaptation Layer" (HAL). HAL se encuentra por encima de HPL y proporciona abstracciones de alto nivel que son más fáciles de usar que HPL pero aún así no proporcionan la funcionalidad completa del hardware subyacente. Los componentes de HAL por lo general tienen un prefijo del nombre del chip. Por ejemplo, el componente de HAL del chip CC2420 es `HalCC2420`.

La parte superior de HAA es "Hardware Independent Layer" (HIL). HIL se encuentra en la parte superior de la capa HAL y proporciona abstracciones que son independientes al hardware. Esta generalización significa que HIL por lo general no proporciona toda la funcionalidad de la capa HAL. Los componentes de HIL no tienen el prefijo en los nombres, ya que representan abstracciones que las aplicaciones pueden utilizar de forma segura y compilar en múltiples plataformas. Por ejemplo, el componente de HIL de la CC2420 en la plataforma MicaZ es

`ActiveMessageC`, que representa la funcionalidad de comunicación de mensajes activos.

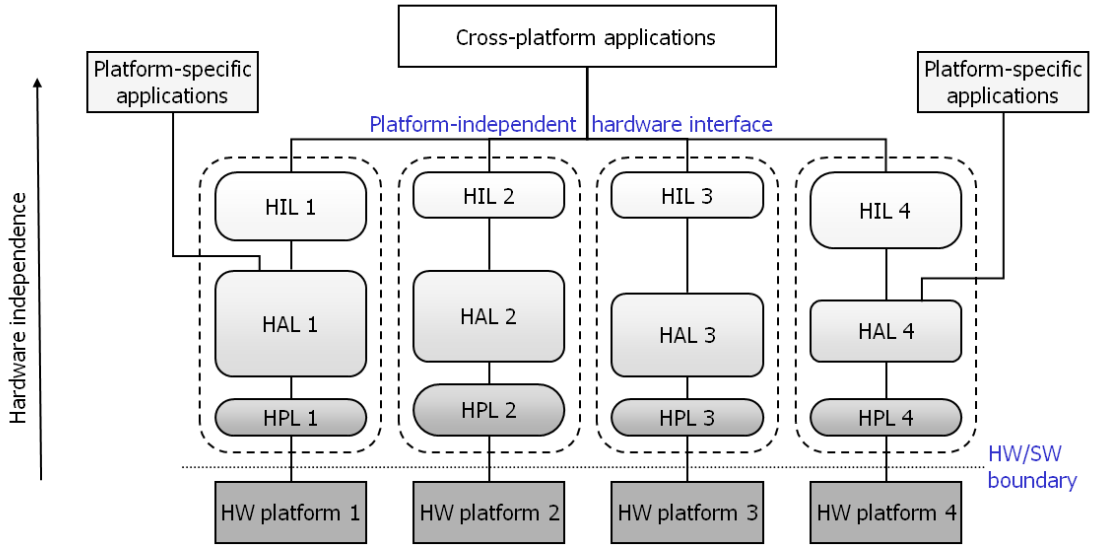


Figura 29: Propuesta de arquitectura de abstracción del hardware

3.2.3.2.2 Planificador

Al igual que con TinyOS 1.x el planificador de TinyOS 2.x tiene una política de FIFO no-preventiva. Sin embargo, las tareas en 2.x funcionan de manera algo diferente que en 1.x.

En TinyOS 1.x, hay una cola de tareas compartidas para todas las tareas, y un componente puede fijar una tarea múltiples veces. Si la cola de tareas está llena, la operación posterior falla. La experiencia con la red muestra que la pila tiene un problema, ya que la tarea podría indicar la finalización de una operación de separación de fase: si el mensaje falla, el componente de arriba se puede bloquear para siempre en espera de la finalización del evento.

En TinyOS 2.x, cada tarea tiene su propio espacio reservado en la cola de tareas, y una tarea sólo puede ser publicada una vez. La publicación falla si y sólo si la tarea ya ha sido publicada. Si un componente necesita enviar una tarea varias veces, se puede establecer una variable de estado interno de forma que cuando la tarea se ejecuta se informa a sí misma. Este ligero cambio en la semántica simplifica en gran medida una gran cantidad de código del componente. Si es preciso el planificador puede sustituir las solicitudes, esto permite a los programadores a probar

nuevas políticas de planificación, como la prioridad o fecha límite base. Pero es importante mantener las políticas no-preventivas, el planificador podría romper con toda la concurrencia en nesC.

3.2.3.2.3 Secuencia de inicio

TinyOS 2.x tiene una secuencia de inicio diferente a la versión 1.x. La interfaz `StdControl` de 1.x ha sido dividida en dos interfaces: `Init` y `StdControl`. Este último sólo tiene dos comandos: `start` and `stop`. En TinyOS 1.x, el cableado (*wiring*) de los componentes para la secuencia de arranque podría hacer que se inicie en el arranque. Este ya no es el caso, la secuencia de arranque sólo inicializa los componentes. Cuando se ha completado la inicialización del planificador, tanto hardware como software, se produce un evento de secuencia de arranque `Boot.booted`, que es cuando el componente de aplicación de alto nivel se encarga de este evento y de los servicios de salida.

3.2.3.2.4 Virtualización

TinyOS 2.x está escrito en nesC 1.2, que introduce el concepto de un componente genérico o componente instanciable. Los módulos genéricos permiten al sistema operativo reutilizar estructuras de datos, tales como vectores de bits y las colas, que simplifican el desarrollo. Además permite una configuración genérica de los servicios para encapsular las relaciones complejas en el cableado (*wiring*) cuando sea necesario.

En la práctica, esto significa que muchos servicios básicos TinyOS están virtualizados. En vez de conectar a un componente con una interfaz de parámetros (por ejemplo, `GenericComm` o `TimerC` en la versión 1.x) un programa instancia un componente de servicio que proporciona la interfaz necesaria. Este componente de servicio hace todo el cableado por debajo (por ejemplo, en el caso de los contadores de tiempo) de forma automática, esto reduce errores de cableado y la simplificación de la abstracción del hardware.

3.2.3.2.5 Temporizadores

TinyOS 2.x proporciona un conjunto mucho más rico de interfaces para los temporizadores (`timer`). La experiencia ha demostrado que los temporizadores

son una de las abstracciones más críticas del sistema. Así en la versión 2.x expande la fidelidad y la forma que toman los temporizadores. Dependiendo de los recursos del hardware de una plataforma, un componente puede utilizar hasta 32KHz, así como temporizadores con fracciones de milisegundos, y el temporizador del sistema puede proporcionar uno o dos temporizadores de alta precisión que se activan de forma asincrónica (tienen la palabra clave async). Los componentes pueden consultar sus temporizadores obteniendo el tiempo antes de que se active y pueda iniciar temporizadores en el futuro (por ejemplo, "empezar a disparar un temporizador en 1Hz a partir de ahora en 31ms). Los temporizadores definen qué componentes del HIL debe proporcionar para cada plataforma. Las plataformas son necesarias para proporcionar temporizadores con fracciones de milisegundo, y pueden proporcionar temporizadores con fracción aun más fina (por ejemplo, 32 kHz), si fuese necesario. Los temporizadores presentar un buen ejemplo de virtualización en TinyOS 2.x. En la versión 1.x la instancia de un temporizador se instancia de la siguiente manera:

```
components App, TimerC;
      App.Timer -> TimerC.Timer[unique("Timer")];
```

En la versión 2.x la instancia de un temporizador es de la siguiente manera:

```
components App, new TimerMilliC();
      App.Timer -> TimerMilliC;
```

3.2.3.2.6 Comunicación

La estructura de mensaje utilizada en TinyOS 2.x es `message_t`. Esta estructura puede albergar cualquier interfaz de comunicación. La estructura en sí es completamente opaca: los componentes no pueden hacer referencia a sus campos. En su lugar, todos los accesos van a través de interfaces. Por ejemplo, para obtener la dirección de destino de un paquete llamado `AM_msg`, un componente llama `AMPacket.destination(msg)`.

Las interfaces de envió distinguen el modo de direccionamiento usado en la comunicación. Por ejemplo, si se utiliza la interfaz de envío `AMSend`, esta requiere una dirección de destino en su llamada. Por el contrario, si la interfaz de envío

utiliza protocolos de *broadcast* o de árbol, las llamadas a la interfaz de envío van libres de direcciones.

La interfaz de mensajes activos `ActiveMessageC` forma parte de la capa HIL (independiente del hardware) de red. Dependiendo de la plataforma usada se define qué interfaz de red es el medio de comunicación a utilizar. Por ejemplo, en la plataforma Mica2 se utiliza la interfaz CC1000 (HAL) como `ActiveMessageC`, mientras que para TMote se utiliza el interfaz CC2420 (HAL) como `ActiveMessageC`.

Ya no se utiliza la dirección `TOS_UART_ADDRESS` para la comunicación de mensajes activos con el puerto serie. En su lugar, se utiliza el componente `SerialActiveMessageC`, que proporciona la comunicación de mensajes activos para el puerto serie.

La comunicación de red usando mensajes activos esta virtualizada a través de cuatro componentes genéricos que reciben el parámetro `AM_type` en su instanciación. Los componentes son `AMSenderC`, `AMReceiverC`, `AMSnooperC` y `AMSnoopingReceiverC`. Al contrario de cómo ocurría en TinyOS 1.x ya no se obtiene error en la llamada a `send()` de `AMSenderC` si otro componente está enviando al mismo tiempo. En cambio, falla si la radio no se encuentra en estado de envío o si el componente `AMSenderC` ya tiene el paquete en pendientes de envío

El sistema de colas de mensajes activos envía los paquetes pendientes. Esto es diferente que el `QueuedSendC` de TinyOS 1.x, en el cual existía una única cola que era compartida por todos los remitentes. Con un número N de componentes `AMSenderC`, hay una única cola donde cada remitente tiene una entrada reservada. Esto significa que cada `AMSenderC` tiene 1/n oportunidades de transmisiones de MAC, donde n es el número de componentes `AMSenderC` con paquetes pendientes de envío. En el peor caso, n es el número de componentes; siempre y cuando cada protocolo y componente que envía paquetes esté intentado enviar paquetes, cada uno recibirá las mismas oportunidades de transmisión.

3.2.3.2.7 Sensores

En TinyOS 2.x, el conjunto de llamadas de los componentes a los sensores comprenden el HIL (capa independiente del hardware) de los sensores de una plataforma. Si un componente necesita frecuencias altas de muestreo, debe utilizar la HAL (capa abstracción del hardware), que da toda la potencia de la plataforma subyacente (plataforma de alta precisión de muestreo independiente no es realmente factible, debido a los datos de las plataformas individuales). El componente `Read` asume que la lectura del sensor puede tolerar algunas latencias. Para evitar grandes latencias se pueden programar las lecturas al sensor mediante una política FIFO.

3.2.3.2.8. Códigos de error

En TinyOS 1.x el código de retorno es `result_t`, cuyo valor puede ser `SUCCESS` (un valor distinto de cero) o `FAIL` (un valor cero). Si bien esto hace que los condicionales en las llamadas sean muy fáciles de escribir (por ejemplo, `if (call Ab ())`), pero no permite al destinatario de la llamada distinguir las causas de error. En TinyOS 2.x, `result_t` se sustituye por `error_t`, cuyos valores incluyen `SUCCESS, FAIL, EBUSY, y ECANCEL`. En los comandos de interfaz y en los eventos se definen los códigos de error que puede devolver y por qué. Desde la perspectiva de migrar el código, éste es una de las diferencias más importantes en 2.x. Por ejemplo:

```
if (call X.y()) {
      busy = TRUE;
}
```

Ahora tienen sus significados invertidos. En 1.x, la declaración de ocupado (`busy`) se ejecutará si la llamada se realiza correctamente, mientras que en 2.x se ejecutará si la llamada falla. Esto anima a un enfoque más portable, actualizable, y de fácil lectura:

```
if (call X.y() != SUCCESS) {
      busy = TRUE;
}
```

3.2.3.2.9 Arbitraje

Mientras que las abstracciones básicas, tales como la comunicación de paquetes y contadores de tiempo pueden ser virtualizados, con la versión 1.x

algunas abstracciones no pueden serlo sin incrementar la complejidad o limitar el sistema. Un ejemplo es el bus compartido del microcontrolador. Muchos sistemas diferentes, como sensores, almacenamiento, radio, tienen que usar el bus al mismo tiempo, por lo que es necesario arbitrar de alguna forma el acceso.

Para apoyar este tipo de abstracciones, TinyOS 2.x presenta una interfaz de recursos que ofrecen una política para arbitrar el acceso al mismo recurso entre varios clientes. Para algunas abstracciones, también se proporciona una política de gestión de energía, ya que puede se decidir cuando el sistema no es necesario y sea desactivado de forma segura.

3.2.3.2.10 Gestión de energía

La gestión de energía en TinyOS 2.x se divide en dos partes: el estado de energía del microcontrolador y el estado de energía de los dispositivos. El primero se calcula en el microprocesador de manera específica mediante el estudio de los dispositivos y las fuentes de interrupción a través de árbitros de recursos. Los servicios totalmente virtualizados tienen sus propias políticas de gestión individual de energía.

TinyOS 2.x proporciona modos de funcionamiento de escucha de baja potencia para las radios CC1000 (Mica2) y CC2420 (Micaz, Telosb, Imote2). Usa para ambas, un enfoque denominado *Low Power Listening* (*LPL*). Los emisores envían preámbulos o envían paquetes varias veces y los receptores despiertan periódicamente para detectar el canal y saber si hay un paquete que se transmite. La radio CC1000, por defecto, utiliza políticas de bajo consumo de energía en la escucha, mientras que para la radio CC2420 se debe configurar en su implementación.

3.2.3.2.11 Protocolos de red

TinyOS 2.x proporciona implementaciones de dos de los protocolos más utilizados en redes de sensores, estos son: difusión (*dissemination*) y recolección (*collection*). Estos dos protocolos permiten una amplia gama de aplicaciones de recopilación y difusión de datos. La recolección de datos ha avanzado

significativamente desde la versión beta más reciente, proporcionando altas tasas de entrega (más de 99%).

3.2.3.2.12 Nombrado de archivos

Como nesC utiliza la misma extensión de archivo `.nc` para todos sus ficheros, tanto para interfaces como para configuraciones y módulos, existe una convención en el nombrado de estos archivo. En la versión 2.x se proponen nuevas convenciones. En la siguiente tabla se muestra el cambio de nombrado de una versión de TinyOS a otra:

Tabla 3: Nuevo nombrado de archivos

	Fichero módulo	Fichero configuración
TinyOS 1.x	`MiAplicacionM.nc`	`MiAplicacion.nc`
TinyOS 2.x	`MiApliacacionC.nc`	`MiAplicacionAppC.nc`

3.2.3.2.13 Conclusión

La versión 2.x representa el siguiente paso del desarrollo de TinyOS. Basándose en las experiencias de los usuarios en los últimos años, ha tomado la arquitectura básica de TinyOS mejorando en varias direcciones, es de esperar que se conduzca al desarrollo de aplicaciones simples y fáciles. Esta nueva versión se encuentra en continuo desarrollo: almacenamiento no volátil, nuevos protocolos de red, y mejoras en las abstracciones de gestión energía, etc.

3.2.4 MySQL

MySQL es un sistema de gestión de base de datos relacional, multihilo y multiusuario. Desarrollado en su mayor parte en ANSI C. MySQL pertenece a Sun Microsystems y ésta a su vez de Oracle Corporation (31). Se trata de una base de datos muy rápida en la lectura, pero puede provocar problemas de integridad en entornos de alta concurrencia en la modificación.

Durante la realización de este proyecto MySQL es utilizado para almacenar en una tabla los datos que la red de sensores inalámbricos recoge para su posterior estudio.

3.2.5 El simulador Avrora

Avrora (32) es un proyecto de investigación del grupo de compiladores de la universidad californiana UCLA. Se trata de un conjunto de herramientas de simulación y análisis de programas escritos para microcontroladores Atmel AVR y utilizados por los nodos sensores Mica2 y MicaZ, entre otros. Avrora proporciona un interfaz en Java y la infraestructura necesaria para la experimentación, elaboración de perfiles y análisis de este tipo de aplicaciones.

La simulación es un paso importante en el ciclo de desarrollo de las aplicaciones para redes de sensores inalámbricas, lo que permite una inspección más detallada de la ejecución dinámica de los programas y de diagnóstico de problemas de software antes de que este se implemente en el hardware.

Avrora también proporciona un marco para el análisis del programa, permitiendo la comprobación estática de software integrado y de una infraestructura para futuros programas de investigación y de análisis. Avrora es flexible, y proporciona una API de Java para el desarrollo eliminando la necesidad de construir una gran estructura de apoyo para el análisis del programa.

3.2.5.1 Características

El nombre de Avrora se deriva del término *aurora borealis*, que significa "el amanecer del norte". El Anexo A Avrora describe detalladamente un manual de instalación del simulador. Entre sus principales características se destacan las siguientes:

- Avrora puede ejecutar una simulación completa de redes de sensores con precisión, permitiendo la comunicación entre los distintos nodos.
- Permite probar una aplicación antes su instalación en el dispositivo hardware. Realizando simulaciones con ciclos de reloj exactos.
- La infraestructura de monitorización de variables permite a los usuarios entender mejor el programa y analizar las posibles optimizaciones.
- Cuenta con utilidades de perfiles que permiten a los usuarios estudiar el comportamiento de su programa durante la simulación.
- Permite la observación detallada del comportamiento del programa sin alterar la simulación, y sin modificar el código fuente del simulador.

- El depurador GDB que permite depuración a nivel y el desarrollo integrado de pruebas.
- La herramienta gráfica de control de flujo puede crear una representación de las instrucciones de su programa, que es útil para entender cómo está estructurado.
- La herramienta de análisis de energía puede analizar el consumo y ayuda a determinar la duración de la batería de su dispositivo.
- Cuenta con una herramienta de control de pila para acotar el tamaño máximo de pila utilizada por el programa.

Capítulo 4 Análisis del problema

En el transcurso de este capítulo se mostrarán las características generales, las particularidades, restricciones y requisitos de problema. Todas estas cuestiones se tendrán en cuenta durante el desarrollo de la/s aplicación/es para la realización de este proyecto.

4.1 Propósito

Se pretende desarrollar un sistema para redes de sensores inalámbricas, que sea capaz de monitorizar la temperatura del entorno en el que se encuentre y además calcular el estado de las baterías de cada nodo sensor. Toda la información producida por la red de sensores debe ser enrutada hasta el gateway, el cual estará conectado vía USB a una estación base que se encargará de almacenar los datos procedentes de la red de sensores en una base de datos para su posterior estudio. La Figura 30 muestra la arquitectura del sistema a desarrollar.

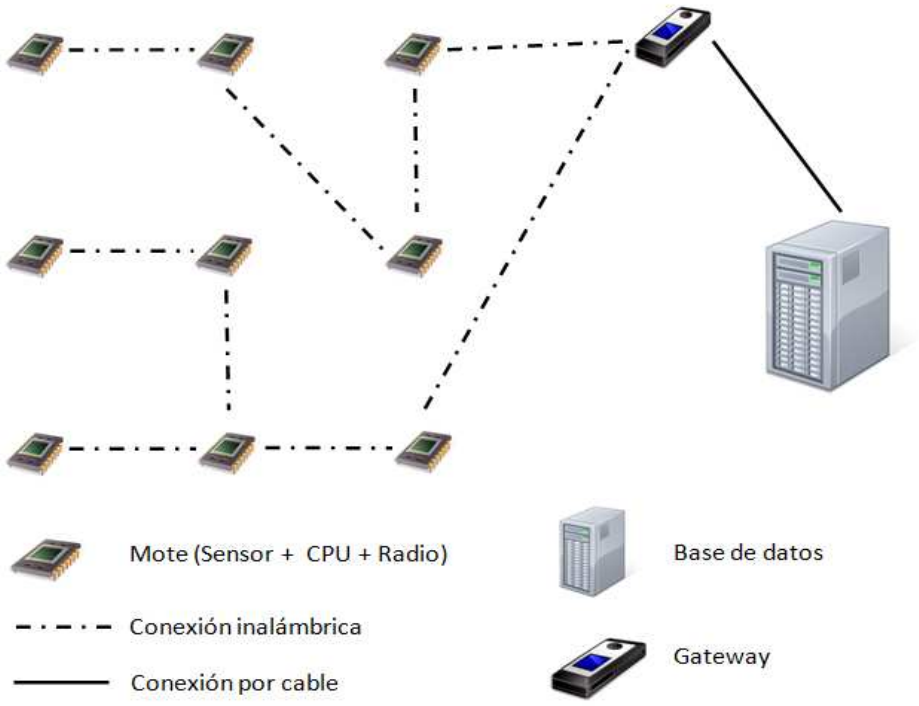


Figura 30: Propuesta de arquitectura para el desarrollo del sistema

4.2 Ámbito del sistema

En esta sección se proporcionará un nombre para el sistema a desarrollar, una breve descripción de lo que el sistema podrá o no podrá hacer y los beneficios de utilizar este sistema.

4.2.1 Propuesta del nombre para el sistema

Se propone como nombre del sistema a desarrollar: ***Sensitive***. Teniendo en cuenta que para cada una de las aplicaciones que componen el sistema recibirá el prefijo ***Sensitive-*** seguido del nombre propio de la aplicación. Las aplicaciones a desarrollar en los distintos componentes del sistema son las siguientes:

- La aplicación que ejecutará en los nodos sensores: *SensitiveSampler.*
- La aplicación que ejecutará en el gateway: *SensitiveBase*
- La aplicación que se encargará de leer los datos provenientes del gateway y almacenarlo en la base de datos: *SensitiveListen.*

4.2.2 Objetivos y beneficios del sistema

Como principal objetivo del sistema se ha fijado la monitorización de la temperatura en el lugar en el que se encuentre desplegada la red de sensores y el análisis de la energía en los nodos. Téngase en cuenta que distintos escenarios podrían ser beneficiados de esta aplicación: desde un edificio hasta un campo de viñedos. Los beneficios de utilizar este sistema para la monitorización de temperatura son inherentes a la utilización de la tecnología WSN, a continuación se enumeran los principales beneficios:

- **Fácil instalación**: los nodos sensores no precisan de ningún tipo de instalación en el medio, basta con colocarlos y se configuran de manera automática para enviar los valores de temperatura obtenidos.
- **Costes de producción reducidos**: la tecnología WSN se ha concebido para que sea barata. Por ello sus dispositivos están diseñados para reducir costes de producción del sistema.

- **Tolerante a fallos**: si en un nodo ocurre un problema parcial o fatal el sistema es capaz de seguir funcionando.
- **Escalable**: se pueden añadir o quitar nodos en el sistema según necesidades sin que ello implique una reestructuración de la red.
- **Gran volumen de información**: el sistema debe ser capaz de generar y almacenar un gran volumen de información obteniendo datos y tendencias más fiables.
- **Sistema en tiempo real**: permite la obtención de datos sobre la temperatura en un instante preciso, o en los intervalos de tiempo que se deseen.
- **Bajo consumo**: el sistema cumple con las restricciones energéticas de las redes de sensores, optimizando el tiempo de vida de cada sensor y por lo tanto de toda la red.
- **Eficiente**: el sistema integra mecanismos que optimizan las características de una WSN. Sin pérdida ni duplicidad de información y encaminando dicha información de manera eficiente hasta el nodo gateway.
- **Respetuoso con el medio ambiente**: la implantación de este sistema no compromete la conservación del medio ambiente. Es por tanto un "sistema verde".

4.2.3 Descripción general del sistema

En esta sección se pasará a describir todos aquellos factores que afectan al producto y a sus requisitos.

4.2.3.1 Perspectiva del sistema

Sensitive es un sistema completo e independiente de otros sistemas, es decir, no está integrado ni forma parte de otro sistema. A continuación se muestra un diagrama de bloques donde se puede observar cómo está compuesto por diferentes aplicaciones y cómo interactúan entre ellos:

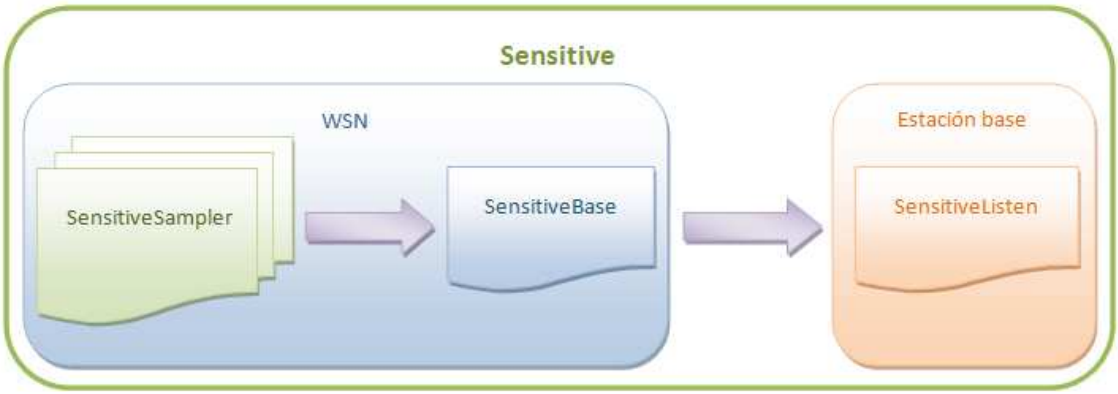


Figura 31: Diagrama de bloques del sistema Sensitive

4.2.3.2 Funcionalidad del sistema

La aplicación que se encuentra ejecutando en la red de sensores obtiene el valor de la temperatura y el voltaje de las baterías de cada nodo una vez transcurrido un intervalo de tiempo (este valor se determina previamente y típicamente será unos pocos segundos). Una vez registrados los valores se envía en un mensaje por radio. Este mensaje viaja por la red hacia el nodo gateway. El nodo gateway obtiene el mensaje y lo envía directamente a la estación base donde el sistema interpretará los valores recibidos y los almacenará en una base de datos.

Para la consecución de la funcionalidad descrita, se puede determinar tres subsistemas o aplicaciones que conformarán el sistema, tal y como muestra la Figura 32:

- **Aplicación para el nodo sensor (*SensitiveSampler*)**: se encargará de obtener la temperatura ambiente y el voltaje de la batería en el nodo sensor y enviarla a la red de sensores. Recibirá el nombre de.
- **Aplicación para el gateway (*SensitiveBase*)**: se encarga de recibir los datos de cada uno de los nodos y lo envía directamente a la estación base.
- **Aplicación para la estación base (*SensitiveListen*)**: recoge los datos que le envía el gateway, los interpreta y los almacena en una base de datos.

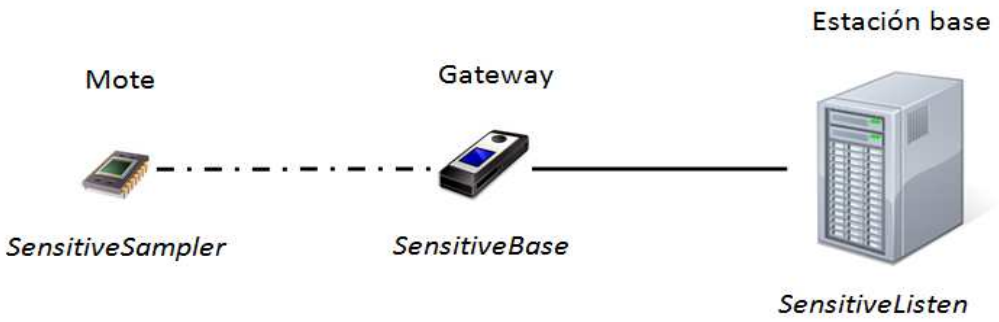


Figura 32: Esquema básico de la funcionalidad del sistema

4.2.3.3 Particularidades y restricciones del sistema

Una particularidad es el sistema en si, por el simple hecho de usar la tecnología WSN para acometer el objetivo de monitorización de la temperatura en un entorno dado. Profundizando se puede destacar:

- La aplicación para los nodos (*SensitiveSampler*) debe minimizar el consumo de energía, dando énfasis al ahorro de energía en la utilización de la radio de cada nodo.
- El sistema tendrá tantos nodos sensores como se desee, únicamente limitado por el número máximo de nodos posibles en la red (255).
- La batería de un nodo sensor consta de 2 pilas tipo AA de 1.5 voltios y 2500mAh cada una de ellas.
- Cada nodo vendrá identificado por un número, el cual es dado cuando la aplicación es instalada en el mote. El nodo gateway se identificará siempre con el número 0.
- La aplicación para el gateway (*SensitiveBase*) es crítica ya que es el nexo de unión entre la red de sensores y la estación base, debe ser eficiente y no causar cuellos de botella en el sistema.
- El sistema tendrá un solo nodo gateway.
- La aplicación para la estación base (*SensitiveListen*) también es crítica. Debe ser capaz de almacenar en una base de datos toda la información que llegue del gateway sin pérdida de información.
- El sistema tendrá una sola estación base y una base de datos donde se almacenará la información de la WSN.

- El formato del mensaje debe ser conocido por todas las aplicaciones del sistema.
- Los datos obtenidos del sistema deben ser almacenados en unidades del Sistema Internacional para la medición de la temperatura (ºC).
- La seguridad de las comunicaciones de la red queda fuera de estudio.

4.2.3.4 Características de los usuarios

El sistema no precisa de la interacción humana para su funcionamiento ya que está totalmente automatizado. Por tanto, no existe la figura de usuario desde la perspectiva de manejo y configuración del sistema, aunque, si existirá la figura de un usuario-consumidor de la información generada y almacenada por el sistema. Este usuario-consumidor utilizará la información de la manera que más le convenga (realización de consultas específicas, generación de informes, profundos estudios, etc.). El perfil del usuario-consumidor deberá ser el de una persona especializada a nivel técnico y capacitado para interpretar los datos recogidos.

4.2.3.5 Suposiciones y dependencias

Se supondrá que durante el desarrollo de este sistema no habrá factores que, si cambian, vayan afectar a los requisitos. Por tanto, no será necesario revisar y cambiar los requisitos.

4.3 Requisitos específicos

En esta sección se expondrán los requisitos referentes a las aplicaciones que se van a desarrollar, aclarando las limitaciones impuestas por el tipo de de desarrollo que se va a acometer.

4.3.1 Requisitos funcionales

El sistema se compone de tres aplicaciones diferentes, por ello se identificarán los requisitos para cada una de las aplicaciones.

4.3.1.1 SensitiveSampler

Número de requisito	RF1

Nombre de requisito	Identificación nodo sensor
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Cada nodo sensor debe tener un identificador único que le sirva para diferenciar a cada uno de ellos, en este caso será un número entero de 1 a N, siendo N mayor que 1 y N menor que 255.

Número de requisito	RF2
Nombre de requisito	Obtención de temperatura
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Cada nodo debe obtener el valor de la temperatura cada cierto intervalo de tiempo.

Número de requisito	RF3
Nombre de requisito	Obtención de voltaje de las baterías
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Cada nodo debe obtener el valor del voltaje de las baterías cada cierto intervalo de tiempo.

Número de requisito	RF4
Nombre de requisito	Envío de datos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	La aplicación, una vez obtenidos los datos de temperatura y voltaje, debe enviarlos por radio hacia el nodo gateway.

Número de requisito	RF5
Nombre de requisito	Multisalto
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Cada nodo, con rol de sensor, debe ser capaz de encaminar hacia el nodo gateway los mensajes que recibe de otro nodo sensor.

Número de requisito	RF6
Nombre de requisito	Obtener mejor enlace
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Cada nodo debe ser capaz de descubrir cada nodo vecino y estimar la calidad de enlace. Determinar la ruta más eficiente para enviar los datos al nodo gateway.

4.3.1.2 SensitiveBase

Número de requisito	RF7
Nombre de requisito	Identificador nodo gateway
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	El nodo gateway debe tener el identificador 0.

Número de requisito	RF8
Nombre de requisito	Puerta de enlace
Tipo	☒ Requisito ☐ Restricción
Fuente del requisito	
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	El gateway debe obtener los datos de de todos los nodos restantes y reenviarlos, por puerto serial, a la estación base.

4.3.1.3 SensitiveListen

Número de requisito	RF9
Nombre de requisito	Transformación de datos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	La aplicación debe ser capaz de leer todos los datos que llegan del puerto serie de la estación base. Y transformarlos en unidades del Sistema Internacional.

Número de requisito	RF10
Nombre de requisito	Almacenamiento de datos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	La aplicación debe ser capaz de almacenar en una base de

	datos los datos recibidos de la red de sensores.

4.3.2 Requisitos no funcionales

Número de requisito	RNF1		
Nombre de requisito	Rendimiento		
Tipo	☒ Requisito	☐ Restricción	
Prioridad del requisito	☐ Esencial	☒ Deseado	☐ Opcional
Descripción	Sistema en tiempo real, sin pérdida ni duplicidad de información. Debe soportar una carga de 2 nodos sensores con un intervalo de monitorización desde 2 a 64 segundos.		

Número de requisito	RNF2		
Nombre de requisito	Fiabilidad		
Tipo	☒ Requisito	☐ Restricción	
Prioridad del requisito	☐ Esencial	☒ Deseado	☐ Opcional
Descripción	El sistema es completamente autónomo de la interacción humana, por tanto debe acercarse a cotas de 100% de fiabilidad		

Número de requisito	RNF3		
Nombre de requisito	Disponibilidad		
Tipo	☒ Requisito	☐ Restricción	
Prioridad del requisito	☐ Esencial	☐ Deseado	☐ Opcional
Descripción	El sistema una vez puesto en marcha debe estar disponible y en funcionamiento. Solo se verá alterado por la duración de las baterías de los nodos sensores.		

Número de requisito	RNF4		
Nombre de requisito	Mantenibilidad de nodos		
Tipo	☒ Requisito	☐ Restricción	
Prioridad del requisito	☒ Esencial	☐ Deseado	☐ Opcional
Descripción	El sistema debe tener un seguimiento, la mantenibilidad del sistema se centra en el recambio de las baterías de los nodos.		

Número de requisito	RNF5
Nombre de requisito	Mantenibilidad de datos

Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Debido al gran volumen de información que proporciona el sistema es preciso mantener los datos. Realizando backups periódicamente.

Número de requisito	RNF6
Nombre de requisito	Plataforma hardware
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Como plataforma hardware, se utilizará el nodo sensor MicaZ, la placa sensora MTS310CB y la placa programadora MIB520 que también realizará funciones de gateway.

Número de requisito	RNF7
Nombre de requisito	Sistema operativo para los nodos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	El Sistema Operativo que se instalará en los nodos será TiinyOS en su versión 2.1.

Número de requisito	RNF8
Nombre de requisito	Sistema operativo para los nodos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	El Sistema Operativo que se instalará en los nodos será TinyOS en su versión 2.1.

Número de requisito	RNF9
Nombre de requisito	Sistema operativo para la estación base
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	El Sistema Operativo que se instalará en la estación base será Ubuntu 9.10, se trata de un SO con licencia GPL, lo que abaratará costes.

Número de requisito	RNF10
Nombre de requisito	Lenguaje de programación para el desarrollo de la aplicación los nodos sensores y gateway
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Se utilizará nesC para el desarrollo de las aplicaciones que se instalarán en los nodos.

Número de requisito	RNF11

Nombre de requisito	Base de datos
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Para almacenar los datos obtenidos de la red de sensores se utilizará el servidor de base de datos MySQL.

Número de requisito	RNF12
Nombre de requisito	Plataforma Java
Tipo	☒ Requisito ☐ Restricción
Prioridad del requisito	☒ Esencial ☐ Deseado ☐ Opcional
Descripción	Será necesaria la instalación de la plataforma de desarrollo Java (JDK y JRE), en su versión 1.6, para el desarrollo y la ejecución del programa que recogerá los datos de la red de sensores e insertarlos en la base de datos.

4.3.3 Interfaces de usuario

La aplicación *SensitiveListen* mostrará los datos recibidos de la red de sensores a través del terminal de la estación base. El interfaz de la misma queda reducido a una interfaz basada en texto donde los datos aparecerán en la consola. Se mostrará, en cada momento, los datos que van llegando provenientes de los nodos sensores, visualizando tanto los datos primitivos (en crudo), como los datos transformados en unidades del Sistema Internacional.

4.3.4 Limitaciones de diseño e implementación

- Para la codificación de *SensitiveSampler* y *SensitiveBase* se utilizará lenguaje nesC, mientras que para *SensitiveListen* se utilizará Java.
- Los nodos sensores y el nodo gateway estarán gobernados por el sistema operativo TinyOS versión 2.1.
- La estación base deberá tener instalado Java versión 1.6 de *Sun Microsystems.*
- La estación base deberá tener instalado el servidor de bases de datos MySQL versión 5.1.37.
- El protocolo de comunicación utilizado viene restringido por las características de una red de sensores inalámbricas. El sistema utilizará el estándar 802.15.4 ZigBee.

Capítulo 5 Diseño de la WSN

Este capítulo pretende mostrar el diseño del sistema basado en el análisis realizado en el capítulo anterior. Además pretende establecer las bases para la implementación del sistema.

5.1 Justificación del software

A continuación se justificarán las decisiones del software utilizado para la realización de este proyecto, valorando el porqué de cada decisión.

5.1.1 Sistema operativo para motes

TinyOS es el sistema operativo elegido para el desarrollo de este proyecto. De entre todos los sistemas operativos existentes en el mercado (véase 2.7 Sistemas Operativos) se opta por TinyOS en su última versión 2.1.1 por las siguientes características:

- Se trata del sistema operativo más extendido para el desarrollo de las WSN.
- Es una plataforma de *código abierto,* lo que supone coste cero.
- Existe una gran comunidad de desarrolladores, lo que facilita la resolución de cualquier duda que se plantee durante el desarrollo.
- Se encuentra en una fase consolidada de desarrollo, lo cual aporta mayor estabilidad y seguridad.

5.1.2 Lenguaje de programación para motes

Asociado al sistema operativo, se encuentra el lenguaje de desarrollo, que en este caso será nesC. Aquí no ha habido flexibilidad alguna, puesto que el propio sistema operativo está construido de manera íntegra en este lenguaje. En el apartado 3.2.1 nesC se puede encontrar más información acerca de este lenguaje.

5.1.3 Sistema operativo para la estación base

Como sistema operativo para la estación base se ha elegido Ubuntu 9.10 *Karmic Koala* (33) por ser un sistema libre y de código abierto, reduciendo así los costes del proyecto. Además en él se integra perfectamente la tecnología Java y el servidor de base de datos MySQL.

5.1.4 Lenguaje de programación para la estación base

Para la implementación de la aplicación que reside en la estación base se ha utilizado el lenguaje orientado a objetos Java. La razón de esta decisión se debe a que TinyOS ofrece multitud de clases Java para el desarrollo de herramientas para el desarrollo sistemas de redes de sensores inalámbricas. Algunas de estas clases serán usadas en la implementación de la aplicación *SensitiveListen.*

5.1.5 Servidor de base de datos

Como servidor de base de datos que reside en la estación base se ha decido utilizar MySQL 5.1.37 (34). La razón de esta elección frente a otros servidores de base de datos se debe a su carácter de software libre. Además por su facilidad de uso a través de las herramientas *MySQL Browser* para la gestión de esquemas y tablas y *MySQL Administrator* para gestión del servidor, creación de *backups* y ejecución de *scripts*.

5.2 Diseño de la aplicación *SensitiveSampler*

En este apartado se ofrece una solución para el diseño de la aplicación que se encargará del sensado, envió y enrutamiento de mensajes de otros nodos para hacerlos llegar hasta el gateway. A continuación se pasará a explicar el comportamiento y la estructura de la aplicación:

5.2.1 Funcionamiento

La aplicación *SensitiveSampler* reside en cada uno de los nodos sensores y tiene como objetivo realizar las siguientes funciones:

- Obtener temperatura del sensor.
- Obtener el voltaje que proporcionan las baterías.

- Configurar el mensaje a enviar con los datos obtenidos.
- Enviar el mensaje a través de la radio del nodo.
- Interceptar para enrutar aquellos mensajes provenientes de otros nodos que lo precisen.

Para realizar todas y cada una de las funciones descritas anteriormente es preciso el uso de un temporizador que indique la frecuencia con la que se van a realizar las mediciones de los sensores. Por motivos de sincronización de la red es necesario que los nodos tengan el mismo intervalo de tiempo para el temporizador. Una vez transcurrido este tiempo el nodo se activa, realiza dichas funciones y una vez concluidas el nodo pasa a un estado de ahorro de energía esperando a que transcurra de nuevo el tiempo para que el temporizador expire.

Por otro lado para que el nodo sensor pueda recibir mensajes de otros nodos es preciso que antes de realizar cualquier acción la radio se ponga en modo escucha (*listening*) Para ello es necesario tener un segundo temporizador para indicar cuanto tiempo antes debe el nodo ponerse a escuchar el medio (véase 5.2.7 Política de ahorro de energía).

El comportamiento final de la aplicación que se ha desarrollado se puede entender muy bien con un diagrama de estados definido en las especificaciones de UML (35). En la siguiente figura se muestra el diagrama de estados de un nodo sensor:

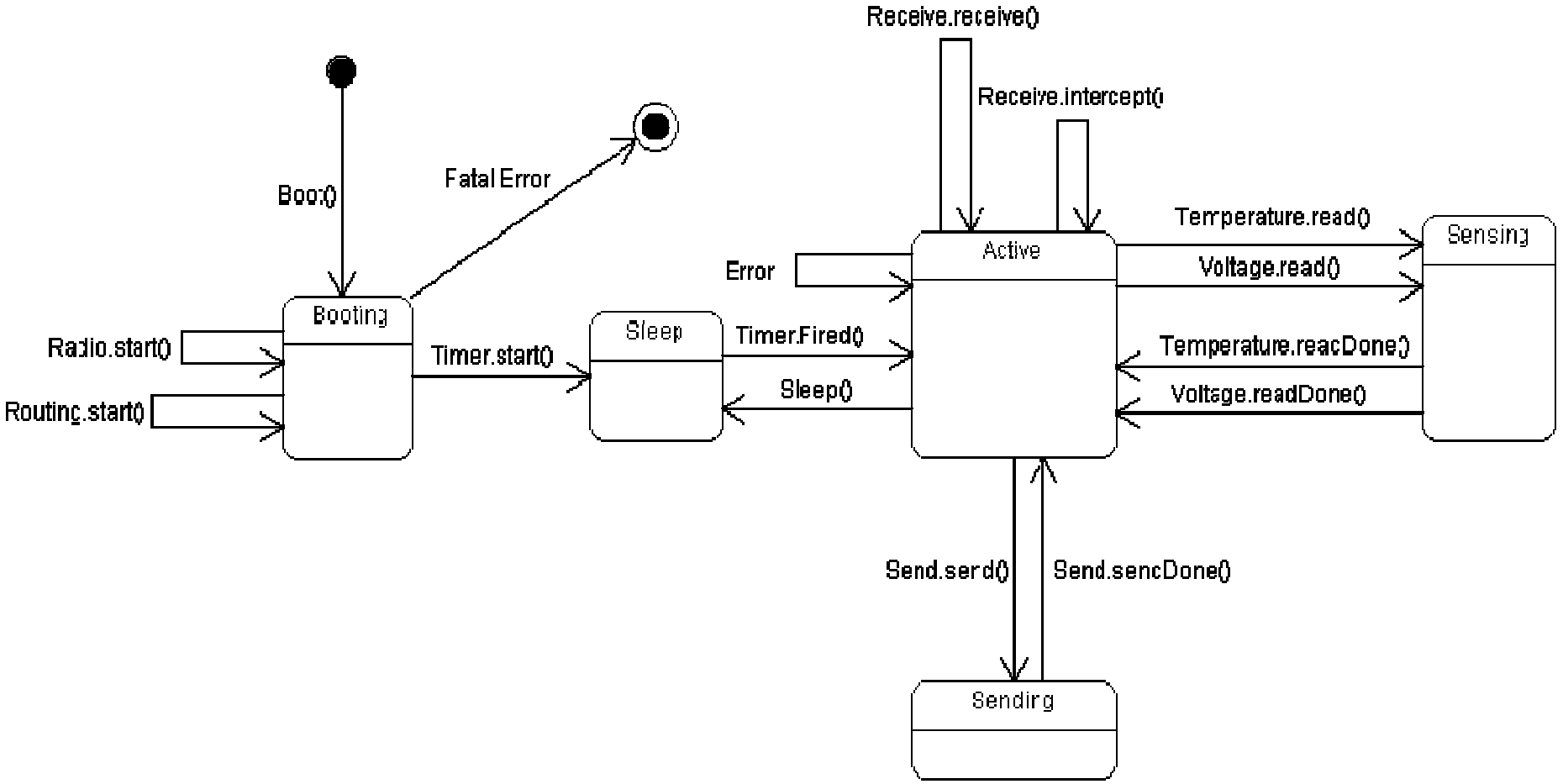


Figura 33: Diagrama de estados de la aplicación *SensitiveSampler*

5.2.2 Estructura

El diseño y desarrollo de aplicaciones en nesC para TinyOS se basa en componentes y en la unión de éstos a través de interfaces. La siguiente figura muestra los componentes de los que hace uso la aplicación *SensitiveSampler*, y la manera en que estos interactúan entre ellos:

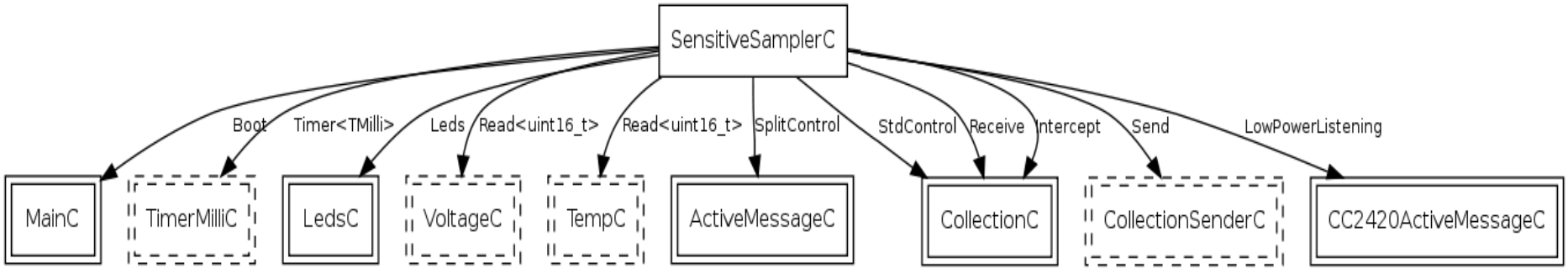


Figura 34: Estructura de la aplicación *SensitiveSampler*

5.2.3 Descripción de los componentes e interfaces

TinyOS ofrece al programador multitud de componentes e interfaces que sirven de base para que el programador implemente sus propios componentes. Por lo general, un nuevo componente es la unión de varios componentes. A continuación se presentan los utilizados en la construcción de la aplicación *SensitiveSampler*, que aparecen en la Figura 34.

5.2.3.1 MainC

Se trata de un componente que dota al componente que lo utiliza la capacidad de actuar como aplicación final. Este componente provee la interfaz `Boot` que indica al componente cuando TinyOS se ha inicializado (inicializa todos sus componentes). Por otra parte utiliza la interfaz `Init` que proporciona el comando `init()` para inicializar el componente. A continuación se muestra el *wiring* del componente:

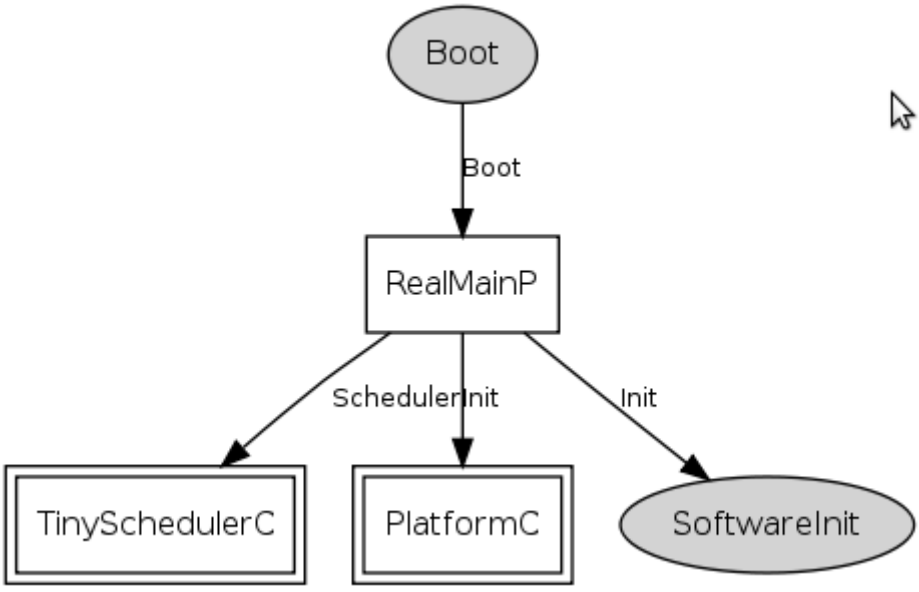


Figura 35: Wiring del componente MainC

5.2.3.2 TimerMilliC

Este componente es una abstracción del temporizador. Para poder usarlo ha de crearse una instancia, con ella se obtiene un temporizador independiente con

precisión de milisegundos. Este componente provee la interfaz `Timer<TMilli>`, proporcionando los siguientes comandos:

- `command uint32_t getdt()`: retorna el tiempo que falta para que el temporizador expire.
- `command uint32_t getNow()`: retorna el tiempo actual del temporizador.
- `command bool isOneShot()`: comprueba si es un temporizador no periódico.
- `command bool isRunning()`: comprueba si el temporizador está funcionando.
- `command void startOneShot(uint32_t dt)`: establece un temporizador simple (no periódico) con `dt` unidades de tiempo.
- `command void startPeriodic(uint32_t dt)`: establece un temporizador periódico a repetir cada `dt` unidades de tiempo.
- `command void stop()`: detiene el temporizador.

Además obliga a implementar el evento:

- `event void fired()`: ocurre cuando el temporizador expira (una vez o periódicamente).

A continuación se muestra el *wiring* del componente:

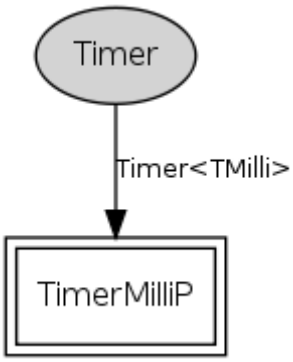


Figura 36: Wiring del componente TimerMilliC

5.2.3.3 LedsC

Los LEDs se controlan a través del componente `LedsC`, Este componente provee la interface `Leds` que proporciona los siguientes comandos:

- `command void led0Off()`: apaga el LED rojo.
- `command void led0On()`: enciende el LED rojo.
- `command void led0Toggle()`: LED rojo. Si está apagado lo enciende. Si está encendido lo apaga.
- `command void led1Off()`: apaga el LED amarillo.
- `command void led1On()`: enciende el LED amarillo.
- `command void led1Toggle()`: LED amarillo. Si está apagado lo enciende. Si está encendido lo apaga.
- `command void led2Off()`: apaga el LED verde.
- `command void led2On()`: enciende el LED verde.
- `command void led2Toggle()`: LED verde. Si está apagado lo enciende. Si está encendido lo apaga.
- `command void set()`: establece la configuración de los LEDs mediante máscara de bits.
- `command void get()`: obtiene la configuración de los LEDs mediante máscara de bits.

A continuación se muestra el *wiring* del componente:

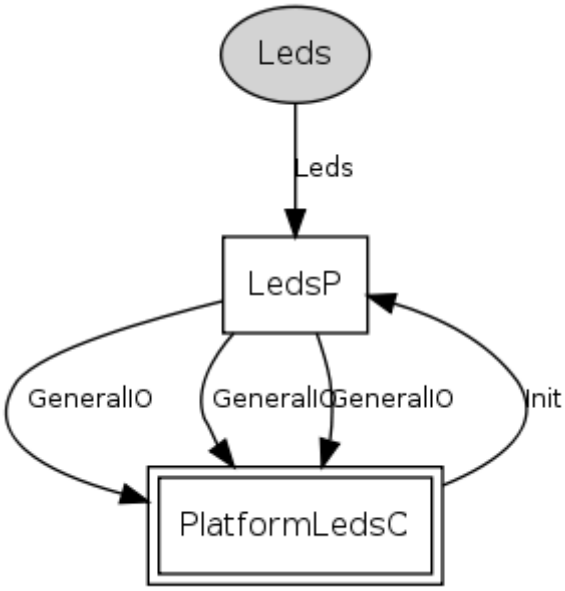


Figura 37: Wiring del componente LedsC

5.2.3.4 VoltageC

Se trata de un componente genérico que por tanto precisa instanciarse. Retorna el valor del voltaje de las baterías. El componente provee la interfaz `Read<uint16_t>`, que presenta el siguiente comando:

- `command error_t read()`: inicia la lectura del valor.

Por otro lado, obliga a implementar:

- `event void readDone(error_t result, uint16_t val)`: ocurre cuando la lectura se ha completado. Retornando si ha ocurrido un error (`error_t result`) y el valor obtenido (`uint16_t val`).

A continuación se muestra el *wiring* del componente:

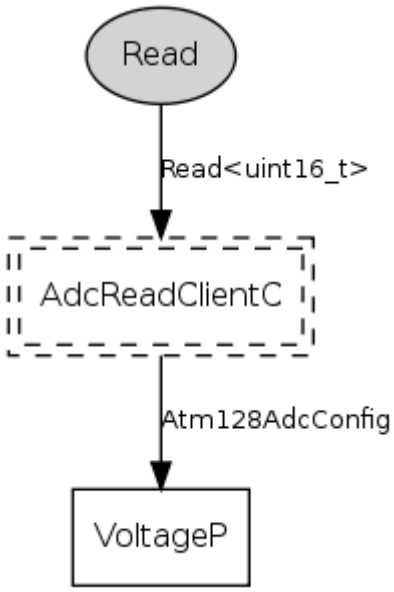


Figura 38: Wiring del componente VoltageC

5.2.3.5 TempC

Al igual que `VoltageC`, se trata de un componente genérico que precisa de instanciación. Para acceder a él, provee la interfaz `Read<uint16_t>`. Este componente nos permite acceder a la placa sensora y obtener el valor de la temperatura.

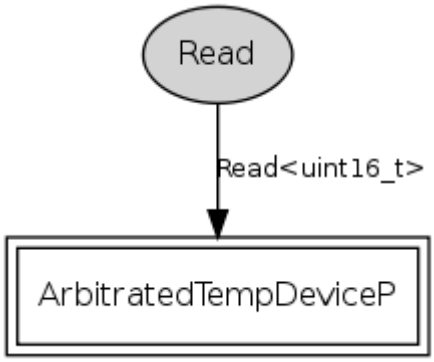


Figura 39: Wiring del componente TempC

5.2.3.6 ActiveMessageC

Este componente implementa la capa de mensajes activos para la plataforma MicaZ. En este caso, se accede a `ActiveMessageC` a través de la interfaz `SplitControl` que se encarga de la conexión para iniciar o parar el componente que provee dicha interfaz. Es decir, se utiliza la interfaz `SplitControl` para iniciar o parar `ActiveMessageC`. Que se realiza utilizando los comandos:

- `command error_t strat()`: inicia el componente y todos sus subcomponentes.
- `command error_t stop()`: para el componentes y todos sus subcomponentes.

Y por otro lado, la utilización de la interfaz obliga a implementar:

- `event void startDone(error_t error)`: notifica que el componente se ha iniciado, con o sin errores, y que está listo en caso de que no los haya.
- `event void stopDone(error_t error)`: notifica que el componente ha sido parado, con o sin errores.

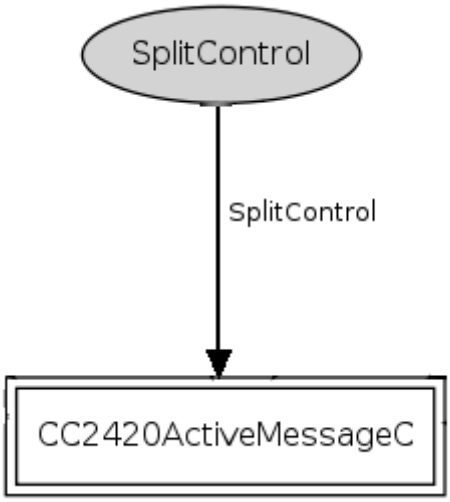


Figura 40: Wiring del componente ActiveMessageC

5.2.3.7 CollectionC

El componente `CollectionC` es uno de los más importantes para el desarrollo de la aplicación ya que presta el servicio de recolección de datos a través de un protocolo de enrutamiento en árbol desde los nodos hoja hasta el nodo raíz o gateway. Este protocolo se denomina *Collection Tree Protocol (CTP)*. Se trata de un componente que ofrece una mayor eficiencia en la recolección de datos de la red, proporciona enrutamiento *multihop* y además permite la existencia de varios nodos raíz. Como puede verse en la siguiente figura, la aplicación accede al componente a través de tres interfaces que se explican a más adelante.

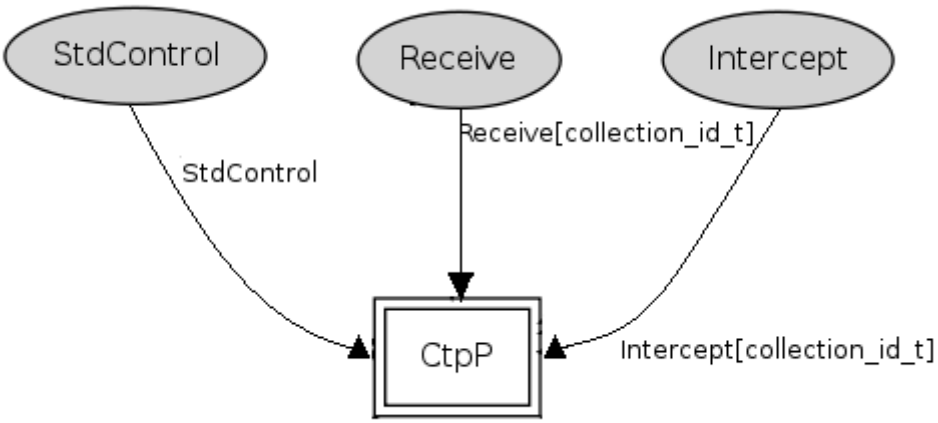


Figura 41: Wiring del componente CollectionC

5.2.3.7.1 StdControl

La interfaz `StdControl` es similar a `SplitControl`, vista anteriormente. Se utiliza para iniciar o parar el componente que provee dicha interfaz, pero su implementación es distinta, como puede verse a continuación:

- `command error_t start()`: inicia el componente y todos sus subcomponentes.
- `command error_t stop()`: para el componente y todos sus subcomponentes.

Se puede observar, que a diferencia de `SplitControl`, `StdControl` no obliga a implementar ningún evento, a cambio los comandos retornan `error_t`, esto implica que la aplicación debe esperar a que el comando termine su tarea.

5.2.3.7.2 Receive

Esta interfaz provee los mecanismos necesarios para recibir mensajes de otros nodos pertenecientes a la red. `Receive` no implementa comandos pero si obliga a implementar el siguiente evento:

- `event message_t * receive(message_t *msg, void *payload, uint8_t len)`: recibe un buffer que contiene el mensaje entrante indicando su tamaño y el valor del mensaje que se retorna puede ser el mismo. Sus parámetros son:
  - `message_t *msg`: paquete del mensaje recibido.
  - `void *payload`: puntero al *payload* (mensaje sin preámbulo).
  - `uint8_t len`: tamaño del *payload*.

Y ha de retornar:

- `message_t *`: buffer de mensaje para próximos mensajes entrantes.

5.2.3.7.3 Intercept

Esta interfaz provee los mecanismos necesarios para recibir un mensaje y reenviarlo a otro destinatario. `Intercept` no implementa comandos pero si obliga a implementar el siguiente evento:

- `event bool forward(message_t *msg, void *payload, uint8_t len)`: recibe un buffer que contiene el mensaje entrante indicando su tamaño y el valor del mensaje que se retorna puede ser el mismo. Sus parámetros son:
    - `message_t *msg`: paquete del mensaje recibido.
    - `void *payload`: puntero al *payload* (mensaje sin preámbulo).
    - `uint8_t len`: tamaño del *payload*.

Y ha de retornar:

- `bool`: `TRUE` indica que el paquete debe ser reenviado, `FALSE` indica que el mensaje no debe ser reenviado.

5.2.3.8 CollectionSenderC

Este componente permite enviar mensajes a la red. Se trata de un componente virtualizado y por tanto ha de instanciarse (a modo de objeto en Java). Para poder acceder al componente provee la interfaz `Send` que ofrece los siguientes comandos:

- `command error_t cancel(message_t *msg)`: cancela la transmisión de un mensaje.
- `command void * getPayload(message_t *msg, uint8_t len)`: retorna un puntero al *payload* del mensaje el cual tiene un tamaño `len`.
- `command uint8_t maxPayloadLength()`: retorna el valor del tamaño máximo del *payload* que la capa de comunicación puede proveer.
- `command error_t send(message_t *msg, uint8_t len)`: envía un mensaje, indicando el tamaño del *payload*.

Además obliga implementar el siguiente evento:

- `event void sendDone(message_t *msg, error_t error)`: ocurre tras finalizar el envío de un mensaje. Sus parámetros son:
    - `message_t *msg`: mensaje a enviar.

- `error_t error`: retorna `SUCCESS` si el envío se ha realizado correctamente, `FAIL` si no y `ECANCEL` si se ha cancelado con el comando `cancel`.

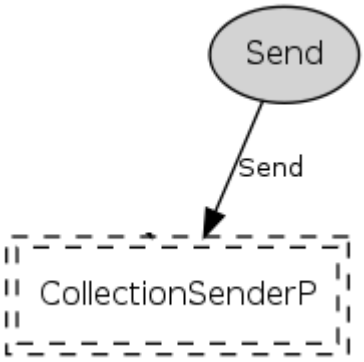


Figura 42: Wiring del componente CollectionSenderC

5.2.3.9 LowPowerListening

La interfaz `LowPowerListening` tiene como principal función minimizar el tiempo que la radio debe escuchar el medio (*listening*) para recibir mensajes de los nodos vecinos. El objetivo es minimizar el consumo de energía debido a la función de *listening*. Para ello debe estar conectado al componente que controla la radio `CC2420ActiveMessageC`. A través de esta interfaz se pueden usar los siguientes comandos:

- `command void setLocalSleepInterval(uint16_t sleepIntervalMs)`: establece en el nodo el intervalo de tiempo que debe apagarse la radio.
- `command void setRxSleepInterval(message_t *msg,uint16_t sleepInterval)`: configura el mensaje saliente para comunicar a los nodos vecinos con el intervalo establecido en el nodo.
- `command uint16_t getLocalSleepInterval()`: retorna el intervalo de LowPowerListening establecido en el nodo.
- `command uint16_t getRxSleepInterval(message_t *msg)`: retorna el valor del intervalo que previamente se ha configurado en el mensaje saliente.

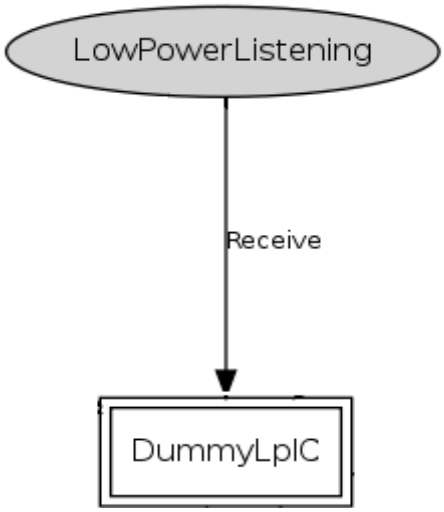


Figura 43: Wiring del componente LowPowerListening

5.2.4 Formato del mensaje

En este apartado se definirá la estructura general del mensaje que se enviará por la red. La estructura del mensaje debe ser conocida por todos los componentes de la red de sensores. Para ello se utilizará una estructura compatible con el estándar 802.14.5 (Zigbee) bajo el paradigma Active Message (AM) que es definido en TinyOS 2.x como `message_t`:

```
typedef nx_struct message_t {
      nx_uint8_t header[sizeof(message_header_t)];
      nx_uint8_t data[TOSH_DATA_LENGTH];
      nx_uint8_t footer[sizeof(message_footer_t)];
      nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

El diseño de `message_t` es general, para facilitar el paso de mensaje entre capas de enlace diferentes. A diferencia de `TOS_Msg` de TinyOS 1.x que es mucho más explícito. Por tanto, cada capa de enlace define sus propias estructuras de cabecera, datos, pie y metadatos (`header`, `data`, `footer` y `metadata`). Para la plataforma MicaZ que usa el chip de radio CC2420 se define su propia estructura de mensaje.

5.2.4.1 Cabecera (header)

La estructura que define la cabecera del mensaje para el chip CC2420 es:

```
typedef nx_struct cc2420_header_t {
      nxle_uint8_t length;
      nxle_uint16_t fcf;
      nxle_uint8_t dsn;
      nxle_uint16_t destpan;
      nxle_uint16_t dest;
      nxle_uint16_t src;
      nxle_uint8_t type;
} cc2420_header_t;
```

El tamaño de la cabecera es de 11 bytes, de ellos se puede destacar:

- `length`: tamaño total del mensaje.
- `dest`: destino del mensaje, en caso de `BROADCAST` la dirección es `0xFFFF`.
- `src`: origen del mensaje.
- `type`: es un número que identifica el tipo de mensaje. Cada aplicación tendrá el suyo propio. Por defecto es `0x7D`.

5.2.4.2 Datos (data)

En el espacio reservado para datos (por defecto 28 bytes) se define la estructura que se implementa en la aplicación a desarrollar. Se trata pues, del espacio de datos que la aplicación debe personalizar para el envío de la información de interés. En particular, para la aplicación *SensitiveSampler* se definirá una estructura que contendrá la siguiente información:

- Node: indica el identificador del nodo origen del mensaje.
- Hops: muestra el número de saltos que el mensaje ha dado por la red hasta llegar al nodo gateway.
- Temperature: contiene el valor de la temperatura obtenido del sensor.
- Voltage: contiene el valor del voltaje de las baterías del nodo.

5.2.4.3 Pie (footer)

La estructura que define el pie del mensaje es:

```
typedef nx_struct cc2420_footer_t {
} cc2420_footer_t;
```

Como puede observarse no hay información definida para el pie de mensaje de la radio CC2420. Si para el desarrollo de una aplicación fuese necesario que el pie almacene algún tipo de información, el desarrollador puede modificar la estructura según sus necesidades.

5.2.4.4 Metadatos (metadata)

La estructura que define los metadatos del mensaje es la siguiente:

```
typedef nx_struct cc2420_metadata_t {
      nx_uint8_t rssi;
      nx_uint8_t lqi;
      nx_uint8_t tx_power;
      nx_bool crc;
      nx_bool ack;
      nx_bool timesync;
      nx_uint32_t timestamp;
      nx_uint16_t rxInterval;
} cc2420_metadata_t;
```

Para el chip CC2420 la estructura de metadatos contiene información extra sobre el mensaje que no será transmitida por radio. La estructura contiene la siguiente información:

- `rssi`: indica la estimación de la conectividad con un nodo y su distancia.
- `lqi`: indica la calidad del enlace con el nodo.
- `tx_power`: informa sobre la potencia de transmisión del nodo.
- `crc`: identifica la integridad del mensaje.
- `ack`: acuse de recibo.
- `timesync`: para la sincronización entre nodos.
- `timestamp`: denota el momento en el que se envía un mensaje.
- `rxInterval`: indica el tiempo en milisegundos que la radio estará activa para recibir mensajes.

5.2.4.5 Paquete CC2420

La estructura definitiva del mensaje de para el chip CC2420 es:

```
typedef nx_struct cc2420_packet_t {
      cc2420_header_t packet;
      nx_uint8_t data[];
} cc2420_packet_t;
```

Puede observarse que la estructura del paquete consta tan solo de la cabecera y del *array* que reserva el espacio necesario para los datos propios de la aplicación. Por ello, no se define en la estructura del pie, ni finalmente se envían los metadatos definidos anteriormente.

La Figura 44 muestra gráficamente el mensaje de TinyOS definido para la aplicación.

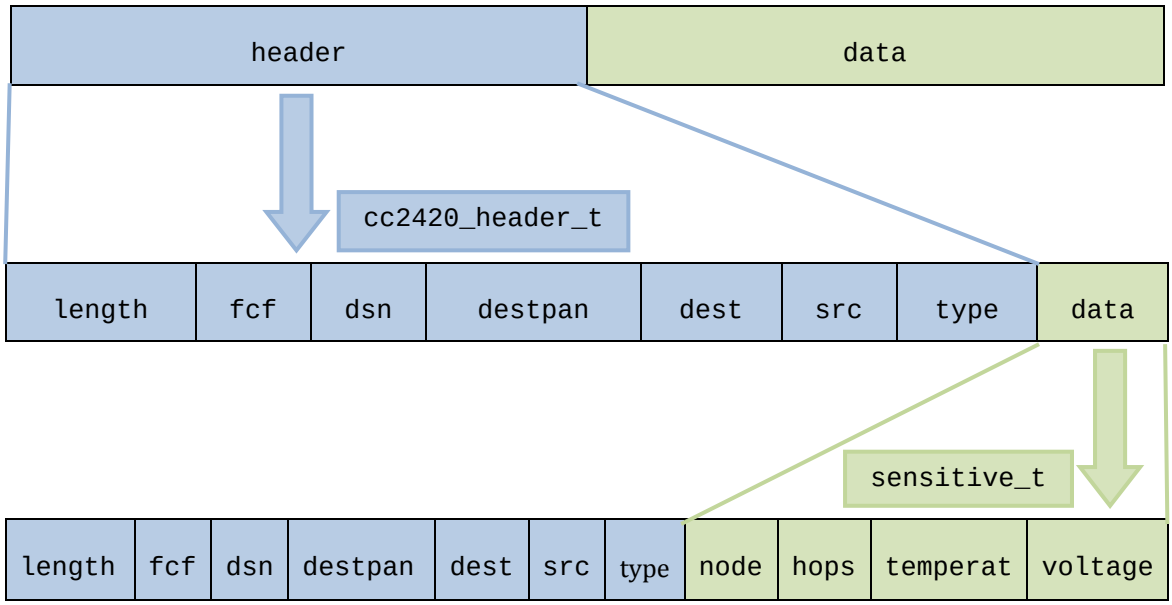


Figura 44: Estructura del paquete de datos para la radio CC2420

5.2.5 Protocolo de estimación de la calidad de enlace (*Link Estimator*)

Los protocolos de enrutamiento requieren de estimadores de enlace (*link estimator*) para descubrir los nodos vecinos. Los nodos pueden estimar la calidad del vínculo con un nodo vecino mediante la proporción de los mensajes enviados con éxito y los mensajes transmitidos en total.

La utilización de *Link Estimator* permite informar a los nodos vecinos sobre su ratio de recepción de paquetes (*packet reception rate, PRR*). Los nodos vecinos reciben estos paquetes y con esa información y su propio *PRR* pueden calcular bidireccionalmente la calidad del enlace.

La calidad del enlace entre un par de nodos (N, M) es la probabilidad de que un paquete transmitido desde N sea recibido satisfactoriamente por el nodo M. La calidad de un enlace bidireccional de un par de nodos indirectos (N, M) es el producto de la calidad del enlace de N a M y de M a N. También es capaz de calcular la calidad del enlace en el caso de que la calidad de N a M y de M a N sea diferente, esto ocurre debido a las posibles interferencias o ruido en el medio.

Puede darse al caso de que un par de nodos (N, M) donde M no puede determinar la calidad de enlace con N, pero N si puede determinar la calidad del enlace con M. Entonces N puede informar a M de la existencia de dicho enlace.

Para informar a un nodo M de la calidad del enlace con el nodo N se crea una tupla (N,q), denominada *Link Information Entry*, donde q es la calidad del enlace del nodo N con M.

El protocolo recibe el nombre de *LEEP* (*Link Estimation Exchange Protocol*) (36) y tiene distintas implementaciones (para este proyecto se utilizará la implementación *4bitle*). Los nodos deben intercambiar la información sobre la calidad de los enlaces mediante la siguiente estructura:

Tabla 4: Estructura del protocolo LEEP

LEEP Header	Payload	LI_Entry 1	LI_Entry 2	…	LI_Entry N

El número de entradas sobre la información del enlace no debe aumentar el tamaño de la estructura LEEP mas allá de la longitud máxima permitida por la capa de enlace de datos (*data link layer*). Puede darse el caso de que la estructura pueda tener 0 entradas `LI_Entry`.

Para calcular la calidad de la conexión bidireccional, *LEEP* mantiene un número de secuencia que se incrementa en uno por cada mensaje saliente. El número de secuencia *LEEP* es usado para contar el número de paquetes que se pierden y así estimar la calidad del enlace. *LEEP* debe transmitir la información que describe las cualidades de enlace de sus nodos vecinos anexando la información de su tabla de vecinos dentro de la capa de enlace de datos, si no hay suficiente espacio para encajar todas las entradas, se retransmitirá en sucesivos

mensajes siguiendo una política *round-robin*. La información recibida permite al nodo receptor determinar la calidad del enlace del nodo transmisor, identificado por la dirección de origen de datos enlace.

5.2.6 Protocolo de enrutamiento

En este apartado se explica el protocolo de enrutamiento que la aplicación utilizará. Para los requisitos expuestos en el capítulo anterior el protocolo que mejor cubre dichas necesidades es *Collection Tree Protocol* (CTP o *Collection*) (37). Se trata de un protocolo de recolección de paquetes de datos, estableciendo enlaces entre los nodos formando una red en forma de árbol. La utilización de CTP como protocolo de enrutamiento garantiza:

- Fiabilidad: ofrece al menos una fiabilidad del 90% de entrega de paquetes, incluso en redes que se encuentren desplegadas en lugares que presenten condiciones difíciles (ruido, obstáculos físicos, etc.). Obtener un 99,9% de fiabilidad en la entrega debe ser factible.
- Robustez: capaz de operar sin reconfigurarse en una amplia gama de condiciones de red, topologías, cargas de trabajo y entornos.
- Eficiencia: la red debe ser capaz de configurarse en uno pocos paquetes de datos. Además es el protocolo que entrega los paquetes desde las hojas hasta la raíz del árbol más eficientemente.

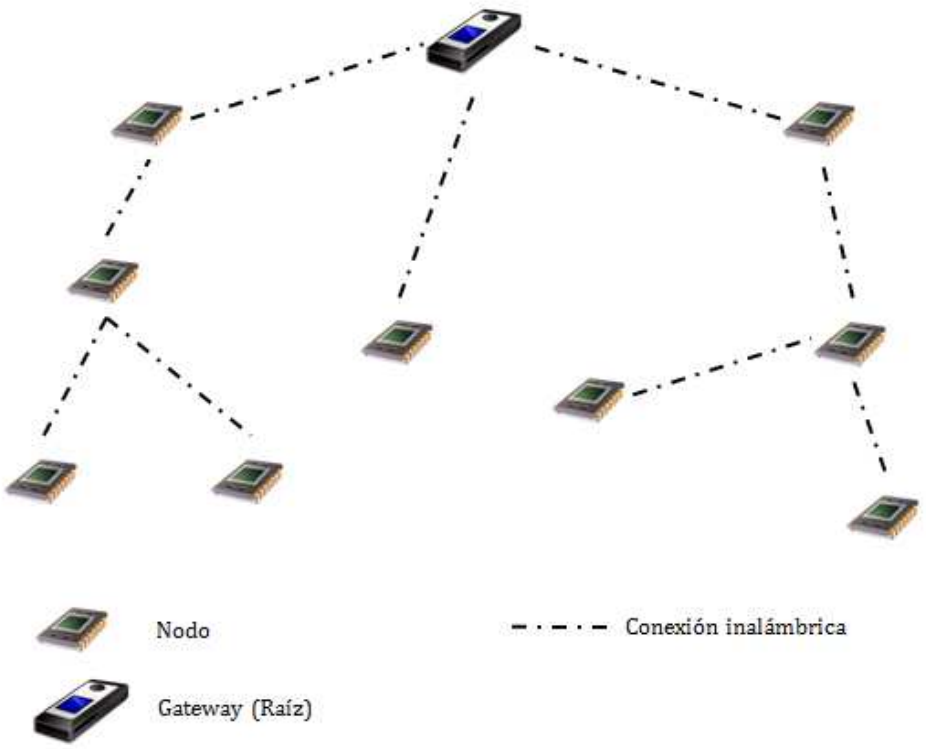


Figura 45: Ejemplo de WSN utilizando Collection Tree Protocol

La Figura 45 muestra un ejemplo de una red de sensores desplegada utilizando *Collection Tree Protocol*. Las WSN son redes muy dinámicas y sufren multitud de cambios en la topología a lo largo de tiempo. Por ello, CTP debe afrontar los siguientes desafíos:

- Detección de bucles, detectar cuando un nodo selecciona uno de sus descendientes como un nuevo padre.
- Supresión de duplicados, detección y tratamiento de pérdidas de paquetes.
- Estimación del enlace, evaluando la calidad del enlace de un nodo con sus vecinos.
- Interferencia, prevención del reenvío de paquetes para prevenir la interferencia con siguientes paquetes.

CTP construye y mantiene arboles eficientes para aquellos nodos que se anuncian como raíces del árbol. Además permite la existencia de varios nodos raíces. La semántica *anycast* entrega por lo menos a una de esas raíces, en particular, a la raíz que suponga un menor coste en términos de enrutamiento. En

otras palabras, *Collection* es un protocolo de libre-dirección (*address-free*), en el cual no se puede especificar explícitamente la dirección de destino del paquete.

Cuando un nodo recibe un mensaje que debe llegar hasta la raíz, éste lo reenvía a otro nodo más cercano a la raíz. Por tanto, los nodos deben ser capaces de inspeccionar los mensajes que reciben antes de reenviarlos, para reunir estadísticas, calcular agregados o suprimir transmisiones redundantes.

Para construir el árbol de enrutamiento CTP utiliza ETX (*expected transmissions*) para determinar el coste de enrutamiento que tiene un nodo con el nodo raíz. Al nodo raíz se le asigna un ETX igual a 0. El ETX del resto de nodos se calcula sumando el ETX del nodo padre más el ETX del enlace con su nodo padre. Dadas una serie de rutas válidas, un nodo, debe elegir la ruta que tenga el menor valor de ETX. Los valores ETX se representan como números reales de punto fijo con precisión de centésimas (16 bit). Por ejemplo, un valor de 451 ETX representa un ETX de 4.51, mientras que un valor de 109 ETX representa un valor de ETX de 1.09.

Los cambios entre los vínculos de los nodos pueden provocar que estos tengan información de la topología obsoleta, lo que puede dar lugar a bucles de enrutamiento y al duplicado de paquetes. Un bucle de enrutamiento ocurre cuando un nodo elige una ruta nueva con un ETX más alto que el anterior, motivado por la posible pérdida de la conectividad con el nodo padre. Si la nueva ruta incluye un nodo que era un descendiente, entonces se produce un bucle. *CTP* detecta bucles a través de dos mecanismos.

En el primer mecanismo (*datapath validation*), cada nodo mantiene y estima su coste de enrutamiento al nodo raíz y agrega este valor al paquete de datos que posteriormente envíe. Cuando un nodo recibe un paquete de un nodo hijo que debe reenviar hacia adelante en el árbol (al nodo padre), este compara el valor ETX del paquete con el suyo propio. Si el valor de ETX recibido es menor, entonces el nodo emisor tiene información obsoleta y un posible bucle de enrutamiento. CTP intenta resolverlo a través del envío *broadcast* de un paquete baliza con la esperanza de que el nodo que envió la trama de datos le escuche y adapte sus rutas en consecuencia.

El segundo mecanismo consiste es no considerar las rutas con un ETX más alto que una constante determinada. El valor de esta constante depende de la implementación.

Por otro lado el duplicado de paquetes es un problema adicional en las redes de sensores inalámbricas. Este fenómeno se produce cuando un nodo recibe una trama de datos con éxito y transmite un ACK, pero el ACK no llega al emisor. El emisor retransmite de nuevo el paquete, y el receptor lo recibe por segunda vez. Esto puede tener efectos desastrosos, debido a que la duplicación es exponencial entre los distintos niveles del árbol de enrutamiento. Por ejemplo, si cada salto se produce un paquete duplicado de promedio, entonces, en el primer salto habrá dos paquetes, en la segundo habrá cuatro, y en el tercero ocho, y así sucesivamente.

Los bucles de enrutamiento pueden dificultar la detección y supresión de duplicados. Si se suprimen los paquetes duplicados de un nodo basándose únicamente en dirección de origen y número de secuencia, puede ocurrir que los paquetes dentro de un bucle sean suprimidos. Para evitarlo, CTP asigna a cada paquete un tiempo vida (THL), que se incrementa en cada salto. Una retransmisión a nivel de enlace tiene el mismo valor de THL, mientras que el paquete que se encuentra en un bucle es improbable que tenga el mismo valor THL.

Si bien los protocolos de recogida pueden tomar una amplia gama de enfoques para abordar estos retos, la interfaz de programación que proporcionan es generalmente independiente de estos detalles. A continuación se describen el conjunto de componentes e interfaces para los servicios de recolección.

5.2.6.1 Interfaces de *Collection*

Un nodo puede tener cuatro roles diferentes en el paradigma de la recolección de datos, productor (*producer*), fisgón (*snooper*), procesador (*processor*) y consumidor (*consumer*). Dependiendo del rol, los nodos usan diferentes interfaces para interactuar con el componente *Collection*:

- Los nodos que generan datos para ser enviados a la raíz son *producers*. Los *producers* utilizan la interfaz Send.

- Los nodos que escuchan otros paquetes de la red son *snoopers*. Los *snooper* utilizan la interfaz `Receive` para recibir mensaje de otros nodos.
- Los nodos que pueden procesar un paquete de la red son *processors*. Los *processors* utilizan la interfaz `Intercept`.
- Los nodos raíz que reciben los datos provenientes de la red son *consumers*. Los *consumers* utilizan la interfaz `Receive`.

Un nodo puede adoptar uno o varios roles al mismo tiempo, es decir, un nodo puede ser productor y procesador, puede enviar paquetes propios y enrutar paquetes de un nodo vecino. También puede darse el caso que un nodo consumidor (raíz, *root*) pueda ser productor. En este caso, el paquete saliente es copiado directamente en el buffer de entrada sin tan siquiera enviarse por radio. Por tanto, el conjunto de todos los nodos raíz y los caminos que conducen a ellos forman la infraestructura de enrutamiento de la red.

5.2.6.2 Servicios de *Collection*

Los servicios del protocolo *Collection* los provee el componente, `CollectionC`, que tiene la siguiente configuración:

```
configuration CollectionC {
      provides {
            interface StdControl;
            interface Send[uint8_t client];
            interface Receive[collection_id_t id];
            interface Receive as Snoop[collection_id_t];
            interface Intercept[collection_id_t id];
            interface RootControl;
            interface Packet;
            interface CollectionPacket;
      }
      uses {
            interface CollectionId[uint8_t client];
      }
}
```

Las interfaces `Receive`, `Snoop` e `Intercept` están parametrizadas por un identificador que corresponde a diferentes modos de operación en la capa superior del protocolo. Todos los paquetes se envían con este identificador para que los nodos *snoopers, processor* y *consumers* puedan analizarlos correctamente.

`CollectionC` solo señala el evento `Receive.receive` en nodos raíz (*root*) cuando un paquete de datos llega correctamente al nodo raíz. Si el nodo raíz realiza una llamada al comando `Send.send` el componente `CollectionC` debe tratarlo como si fuera un paquete recibido, es decir, copia el paquete de datos que se encuentra en el buffer de salida en el buffer de entrada y señala el evento `Receive.receive`. Sobre el comportamiento de `CollectionC` destaca:

- Si `CollectionC` recibe un paquete de datos para reenviarlo al siguiente nodo en la ruta hasta el nodo raíz debe señalar el evento `Intercept.forward`.
- Si `CollectionC` recibe un paquete de que debe ir a un nodo diferente debe señalar el evento `Snoop.receive`.
- La interfaz `RootControl` permite a un nodo establecerse como nodo raíz del árbol. `CollectionC` no debe configurar, por defecto, un nodo como raíz (*root*).
- Las interfaces `Packet` y `CollectionPacket` permiten al componente que lo use acceder los campos del paquete de datos.

5.2.6.2.1 Componente *CollectionSenderC*

Se trata de un componente virtualizado y que precisa de instanciación. Tiene la siguiente configuración:

```
generic configuration CollectionSenderC(collection_id_t collectid)
{
      provides {
            interface Send;
            interface Packet;
      }
}
```

Este componente permite el envió de paquetes de datos y además permite el acceso a los campos del paquete para procesar el mensaje, si es necesario.

5.2.6.3 Implementación

La implementación del protocolo *Collection* (CTP) consta de tres principales componentes:

- *Link estimator*, es responsable de la estimación de la calidad del enlace de un nodo con sus vecinos.
- *Routing engine,* usa las estimaciones de enlace e información sobre la red para decidir que vecino es el siguiente en la ruta.
- *Forwarding engine,* mantiene la cola de paquetes de datos que serán enviados. Decide si enviarlos y cuando.

5.2.6.3.1 Link estimator

La implementación de *Link Estimator* utilizada en este proyecto es 4bitle y se encuentra codificada en el componente `LinkEstimatorP`. *Link Estimator* sigue las pautas del protocolo LEEP explicadas anteriormente (véase 5.2.5 Estimador de enlace de la calidad de enlace (Link Estimator)). La estimación de la calidad del enlace se desvincula de la creación de rutas y el único requisito es que la calidad resultante esté estandarizada. La radio proporciona valores como LQI, RSSI y estimaciones de la calidad del enlace para calcular ETX (*Expected Transmision*), o combinaciones de estos para realizar estimaciones. El motor de enrutamiento instruye al componente `LinkEstimatorP` para insertar la existencia de un nodo en la tabla del nodo vecino, a través del cual conseguir una ruta de calidad hasta la raíz dentro de la tabla de enrutamiento del nodo vecino.

`LinkEstimatorP` puede enviar mensajes de control propios para calcular la calidad del enlace con otro nodo en ambos sentidos. Para ello ofrece servicios para actualizar las estimaciones basadas en la transmisión de datos a los nodos vecinos.

`LinkEstimatorP` utiliza la interfaz `LinkPacketMetadata` para determinar si el canal es de gran calidad cuando se recibe un paquete de un nodo vecino a considerar el vínculo y la inserción en la tabla de enrutamiento del vecino.

Con `LinkEstimatorP` se puede insertar manualmente la existencia de enlace con un nodo en la tabla de enrutamiento para prevenir que un nodo quede aislado del resto de la red.

5.2.6.3.2 Routing Engine

El componente `CtpRoutingEngineP` contiene la implementación de *Routing Engine*. Es responsable del cómputo de las rutas hacia la raíz o raíces del árbol. Es parte del plano de control de la red, pero no se encarga de la retransmisión de paquetes de datos pertenecientes a otros nodos.

`CtpRoutingEngineP` utiliza la interfaz *LinkEstimator* para conocer los nodos de la tabla de vecinos mantenida por `LinkEstimatorP` y la calidad de dichos enlaces en ambos sentidos. `CtpRoutingEngineP` permite que un nodo sea configurado como raíz dinámicamente. Además mantiene el enlace con múltiples nodos para enviar paquetes de datos y evitar la pérdida de paquetes.

5.2.6.5.3 Forwarding Engine

El componente que implementa *Forwarding Engine* es `CtpForwardingEngineP`. Este componente proporciona todas la interfaces de alto nivel (excepto *RootControl*) de las cuales *Collection* provee a la aplicación que lo utiliza. Sus principales funciones son: calcular posibles retransmisiones, supresión de duplicados, detección de bucles e informar a *Link Estimator* del resultado de las retransmisiones que se han realizado.

`CtpForwardingEngineP` utiliza un gran número de interfaces, las cuales pueden ser divididas en grupos según su funcionalidad:

- Comunicación de un solo salto (*single hop*), transmitiendo los paquetes al siguiente nodo e informando sobre la estimación del enlace.
- Enrutamiento, decidiendo cuando transmitir un paquete al siguiente nodo de la ruta. Detectando rutas inconsistentes e informándolo a *Routing Engine*.
- Mantenimiento de buffer y cola de paquetes a transmitir, los cuales pueden ser paquetes propios o de otros nodos que han de retransmitirse.
- Control del tiempo de vida de un paquete, detectando duplicados de paquetes causados por la pérdida de ACK's.

5.2.7 Política de ahorro de energía.

TinyOS proporciona una herramienta para controlar el consumo de energía que realiza la radio a través de un mecanismo conocido como *Low Power Listening* (LPL). Esta técnica permite encender la radio el tiempo necesario para detectar posibles paquetes en el canal. Si se detecta, mantiene la radio encendida hasta recibir el paquete completo. El periodo de comprobación de LPL debe ser mayor que el tiempo necesario para detectar un paquete, un nodo emisor debe enviar el primer paquete con tiempo suficiente para que el nodo receptor pueda escucharlo. El emisor estará activo hasta que reciba un acuse de recibo (ACK) o haya concluido el *timeout.* El *timeout* es unas milésimas de segundo más largo que el periodo de comprobación del receptor. Cuando un nodo recibe un paquete, pasa a estar a la espera el tiempo suficiente para recibir un segundo paquete. Debido a que una ráfaga de paquetes amortiza el coste de recibir el primer paquete. Por tanto, es más eficiente enviar paquetes a ráfagas cuando se usa LPL que enviar paquetes individuales con una tasa constate de envío.

Para desarrollar aplicaciones que permitan el uso eficiente de la radio, TinyOS proporciona la interfaz `LowPowerListening`. Esta interfaz es soportada por las radios CC1000, CC2420 y RF230.

```
interface LowPowerListening {
 /**
  * Set this this node's radio wakeup interval, in milliseconds. After
  * each interval, the node will wakeup and check for radio activity.
  *
  * Note: The wakeup interval can be set to 0 to indicate that the radio
  * should stay on all the time but in order to get a startDone this
  * should only be done when the duty-cycling is off (after a stopDone).
  *
  * @param intervalMs the length of this node's Rx check interval, in [ms]
  */
 command void setLocalWakeupInterval(uint16_t intervalMs);
 /**
  * @return the local node's wakeup interval, in [ms]
  */
 command uint16_t getLocalWakeupInterval();
 /**
  * Configure this outgoing message so it can be transmitted to a neighbor
mote
  * with the specified wakeup interval.
  * @param 'message_t* ONE msg' Pointer to the message that will be sent
  * @param intervalMs The receiving node's wakeup interval, in [ms]
  */
 command void setRemoteWakeupInterval(message_t *msg, uint16_t intervalMs);
 /**
  * @param 'message_t* ONE msg'
  * @return the destination node's wakeup interval configured in this
message
  */
 command uint16_t getRemoteWakeupInterval(message_t *msg);
}
```

El uso de LPL implica, en primer lugar el establecimiento de un ciclo de trabajo en el evento `Boot.booted()` de la aplicación a desarrollar. Por cada paquete que la aplicación desea enviar, debe configurarlo indicando el ciclo de trabajo designado, y así, calcular el número correcto de preámbulos a enviar. A continuación se muestra un ejemplo de uso:

```
event void Boot.booted() {
      call LPL.setLocalSleepInterval(LPL_INTERVAL);
      call AMControl.start();
}

event void AMControl.startDone(error_t e) {
      if(e != SUCCESS)
      call AMControl.start();
}

…

void sendMsg() {
      call LPL.setRxSleepInterval(&msg, LPL_INTERVAL);
      if(call Send.send(dest_addr, &msg, sizeof(my_msg_t))!=
      SUCCESS)
            post retrySendTask();
}
```

Como puede observarse en el código anterior, antes de iniciarse la radio debe establecerse el intervalo que dormirá la radio en cada ciclo. Esto se realiza a través del comando `setLocalSleepInterval()`. La llamada al comando `setRxSleepInterval()` configura el mensaje a enviar para que la radio envié el número correctos de preámbulos.

5.3 Diseño de la aplicación *SensitiveBase*

En este apartado se ofrece una solución para el diseño de la aplicación que se encargará de recibir todos los mensajes provenientes de la red y reenviarlos a la estación base. A continuación se pasará a explicar el comportamiento y la estructura de la aplicación:

5.3.1 Funcionamiento

La aplicación *SensitiveBase* tiene como objetivo realizar las siguientes funciones:

- Establecerse como nodo raíz (*root*) dentro de la red de sensores.
- Recibir correctamente todos los paquetes de datos provenientes de los nodos sensores.
- Enviar todos los paquetes de datos, recibidos de los nodos sensores, vía puerto serial a la estación base.

El funcionamiento de *SensitiveBase* es totalmente distinto del de *SensitiveSampler*. No realiza funciones de obtención de temperatura ni de voltaje, no realizará funciones de enrutamiento para otros nodos, ya que todas las rutas desembocan en él (nodo raíz). Será preciso el manejo de un buffer que almacene los paquetes de datos entrantes evitando situaciones de cuello de botella sin perder información. Por otro lado, el nodo raíz forma parte fundamental de la red y tomará el rol de *Root* que proporciona la interfaz `RootControl` de *Collection Tree Protocol*. Como el gateway (raíz) está conectado mediante cable USB a la estación base no precisa la implementación de una política de ahorro de energía (LPL).

El comportamiento final de la aplicación que se ha desarrollado se puede entender muy bien con un diagrama de estados definido usando la notación de UML. En la siguiente figura se muestra el diagrama de estados del nodo raíz:

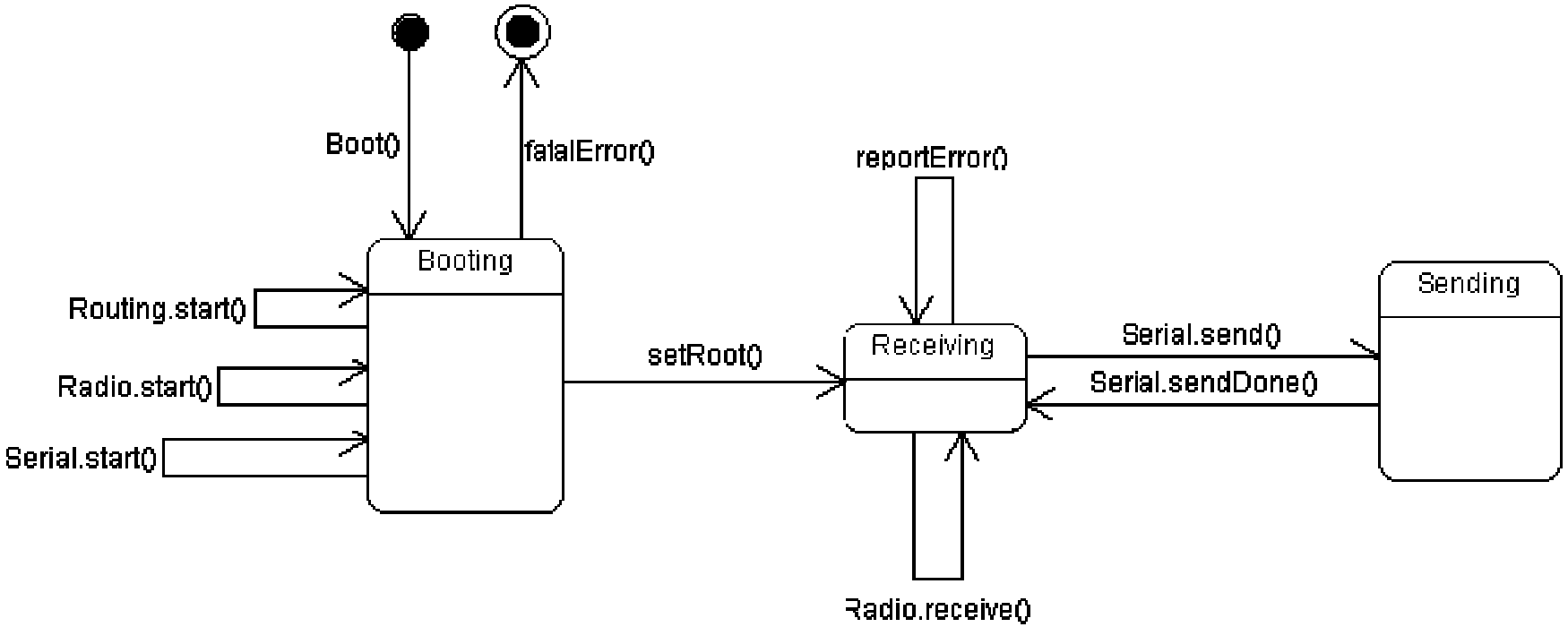


Figura 46: Diagrama de estados de la aplicación *SensitiveBase*

5.3.2 Estructura

Al igual que en la aplicación para los nodos sensores, el diseño y desarrollo de la aplicación para el nodo raíz se basa en componentes y en la unión de éstos a través de interfaces. La siguiente figura muestra los componentes de los que hace uso la aplicación *SensitiveBase*, y la manera en que estos interactúan entre ellos:

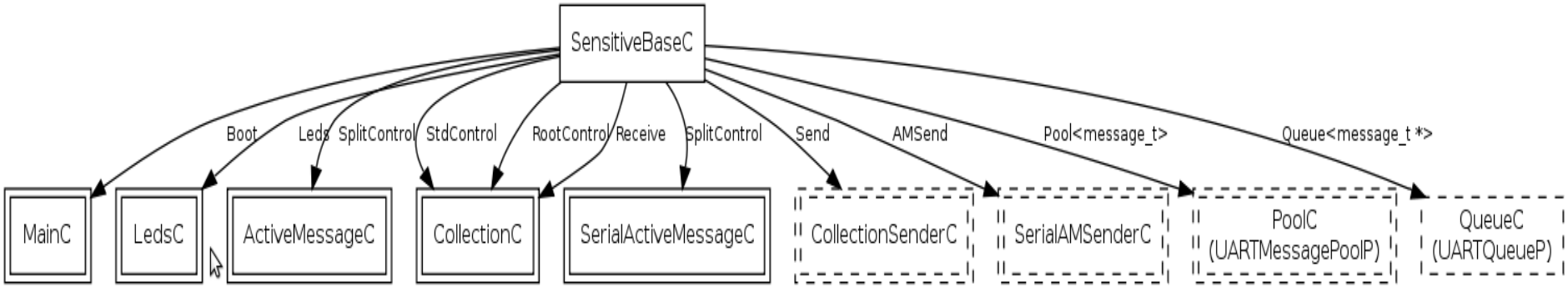


Figura 47: Estructura de la aplicación *SensitiveBase*

5.3.3 Descripción de componentes e interfaces

A continuación se presentan las interfaces y componentes utilizados en la construcción de la aplicación *SensitiveBase*. Algunos de los componentes utilizados coinciden con los ya explicados anteriormente en *SensitiveSampler*. Solo se explicarán aquellos específicos de la aplicación para el nodo raíz.

5.3.3.1 CollectionC

Aunque este componente es utilizado en *SensitiveSampler* la forma de acceder por parte de *SensitiveBase* es distinta, obteniendo así, un comportamiento totalmente distinto. Utiliza las interfaces `RootControl` para establecerse cono nodo raíz dentro de la red y la interfaz `Receive` para recibir los mensajes provenientes de los nodos sensores. Además utiliza la interfaz `StdControl` para iniciar y parar el componente. Como puede verse en la siguiente figura la aplicación accede al componente a través de tres interfaces.

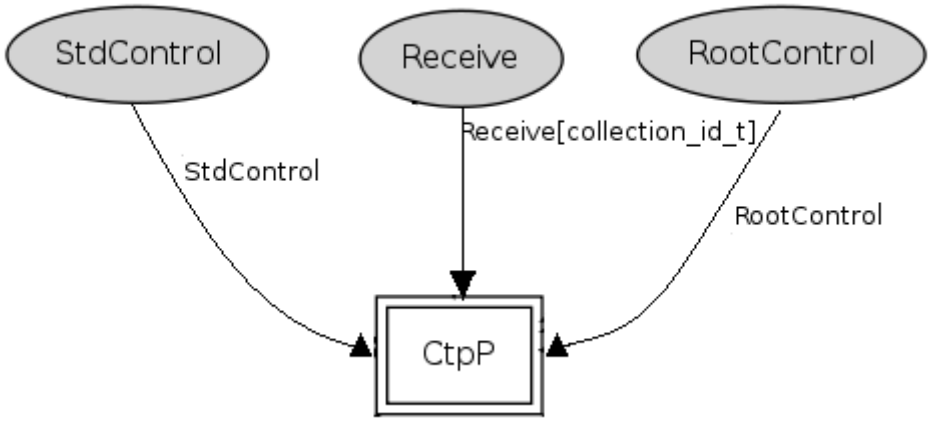


Figura 48: Wiring del componente CollectionC

5.3.3.1.1 RootControl

Esta interfaz proporciona los mecanismos necesarios para establecer a un nodo como raíz dentro del protocolo de enrutamiento. Proporciona los siguientes comandos:

- `command bool isRoot()`: pregunta si el nodo está establecido como raíz. Retorna `TRUE` si es raíz, si no, `FALSE`.

- `command error_t setRoot()`: establece al nodo como raíz.
- `command error_t unsetRoot()`: deshabilita el modo *root* al nodo.

5.3.3.2 SerialAMSender

Se trata del componente encargado del envió de los paquetes de datos recibidos vía serial. Se trata de un componente genérico parametrizado que precisa de instanciación. Tiene el siguiente *wiring*:

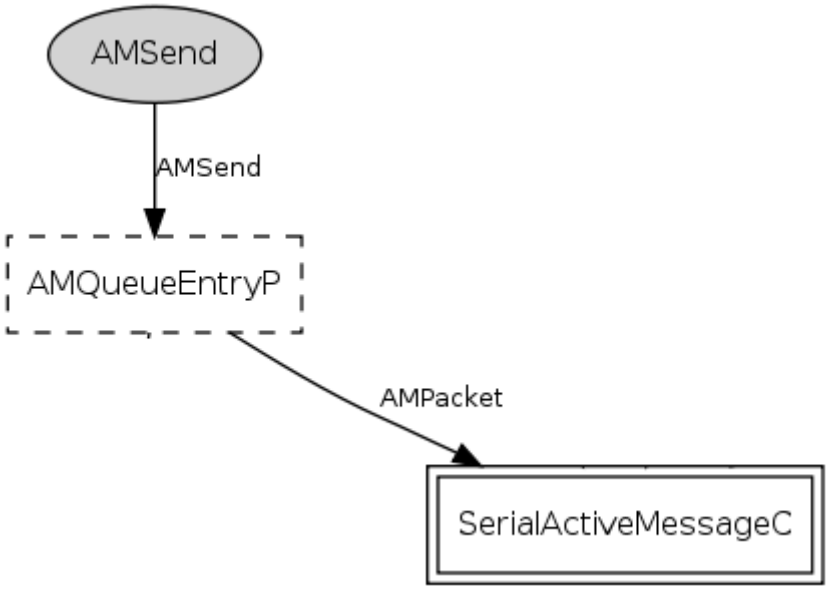


Figura 49: Wiring del componente SerialAMSender

El componente provee cuatro interfaces:

- `PacketAcknowledgements`: permite al componente establecer o deshabilitar los acuses de recibo por paquete.
- `AMPacket`: permite acceder a los paquetes.
- `AMSend`: se encarga del envío mensajes.
- `Packet`: permite el acceso a la información del paquete de datos.

SensitiveBase utiliza el componente `SerialAMSender` a través de la interfaz `AMSend`, el cual provee los siguientes comandos:

- `command error_t cancel(message_t *msg)`: cancela la transmisión del paquete de datos en curso.

- `command void * getPayload(message_t *msg, uint8_t len)`: retorna un puntero que apunta a la región de datos del paquete.
- `command uint8_t maxPayloadLength()`: retorna el tamaño máximo de la región de datos que la capa de comunicación puede proporcionar.
- `command error_t send(am_addr_t addr, message_t *meg, uint8_t len)`: envía un paquete indicando el tamaño de los datos, a la dirección indicada.

Por otro lado, ha de implementarse el siguiente evento:

- `event void sendDone(message_t * msg, error_t error)`: ocurre cuando la llamada a `send()` ha terminado.

5.3.3.3 PoolC

Este componente proporciona un buffer de memoria dinámica, se utiliza para el almacenamiento intermedio de los paquetes de datos entrantes y así evitar situaciones de cuello de botella. Se accede a este componente a través de la interfaz `Pool<pool_t>`. El *wiring* del componente es:

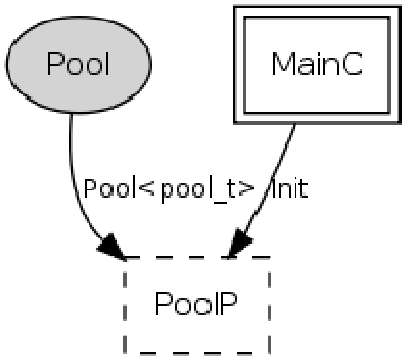


Figura 50: Wiring del componente PoolC

La interfaz `Pool<pool_t>` es un espacio de memoria asignado a un tipo de objetos específicos indicados por parámetro. La interfaz permite alojar y desalojar estos elementos. Implementa los siguientes comandos:

- `command bool empty()`: retorna `TRUE` cuando no contiene ningún elemento. `FALSE` si tiene al menos uno.
- `command t * get()`: aloja un elemento.

- `command uint8_t maxSize()`: retorna el número máximo de elementos.
- `command error_t put(t *newVal)`: desaloja un objeto.
- `command uint8_t size()`: retorna el número de elemento elementos que tiene en ese momento.

5.3.3.4 QueueC

Este componente proporciona una cola FIFO con tamaño limitado de elementos. El componente provee la interfaz `Queue<typedef_t>` que maneja los elementos que contiene la cola. Implementa los siguientes comandos:

- `command t dequeue()`: elimina el primer elemento de la cola.
- `command bool empty()`: retorna `TRUE` si la cola está vacía. `FALSE` si contiene algún elemento.
- `command error_t enqueue(t newVal)`: encola un elemento al final de la cola.
- `command t head()`: obtiene la cabeza de la cola sin eliminarla de la misma.
- `command uint8_t maxSize()`: retorna el número máximo de elementos que la cola puede almacenar.
- `command uint8_t size()`: retorna el número de elementos que contiene la cola en ese momento.

5.3.4 Formato del mensaje serial

La aplicación maneja dos formatos distintos de mensaje debido a que utiliza dos protocolos de comunicación distintos. El primero, comunicación vía radio entre los nodos sensores, previamente descrito en 5.2.4 Formato del mensaje. Y el segundo, comunicación vía serial entre el nodo raíz (gateway) y la estación base. En este apartado se explicará en detalle el segundo.

La utilización de una estructura de mensaje distinta se debe a que el envió de datos desde el gateway hasta la estación base precisa de un protocolo de comunicación más sencillo. La estructura que se enviará es la siguiente:

```
typedef nx_struct serial_packet {
      serial_header_t header;
      nx_uint8_t data[];
} serial_packet_t;
```

El paquete serial consta de una cabecera y de un *array* de bytes reservado para los datos propios de la aplicación, cuyo tamaño no debe ser mayor de 28 bytes (definido en la constante TOS_DATA_LENGTH). Estos datos son los explicados en el apartado 5.2.4.2 Datos (data). A continuación se explica la información contenida en la cabecera `serial_header_t`.

5.3.4.1 Cabecera (Serial header)

La estructura de datos de la cabecera del paquete serial tiene la siguiente forma (definida en `tinyos2.x/tos/lib/serial/Serial.h`):

```
typedef nx_struct serial_header {
      nx_am_addr_t dest;
      nx_am_addr_t src;
      nx_uint8_t length;
      nx_am_group_t group;
      nx_am_id_t type;
} serial_header_t;
```

Como puede observarse la estructura contiene la siguiente información:

- `dest`: dirección destino del paquete de datos (2 byte).
- `src`: dirección origen del paquete (2 byte).
- `length`: tamaño en bytes del *payload* del paquete (1 byte).
- `group`: identificador que sirve para identificar distintos grupos de trabajo dentro de la misma red (1 byte).
- `type`: es un número que identifica el tipo de mensaje. Cada aplicación tendrá el suyo propio (1 byte).

El tamaño de la cabecera será de 7 bytes a diferencia de los 11 bytes de la cabecera del paquete CC2420. A continuación se muestra la figura con la estructura del paquete serial.

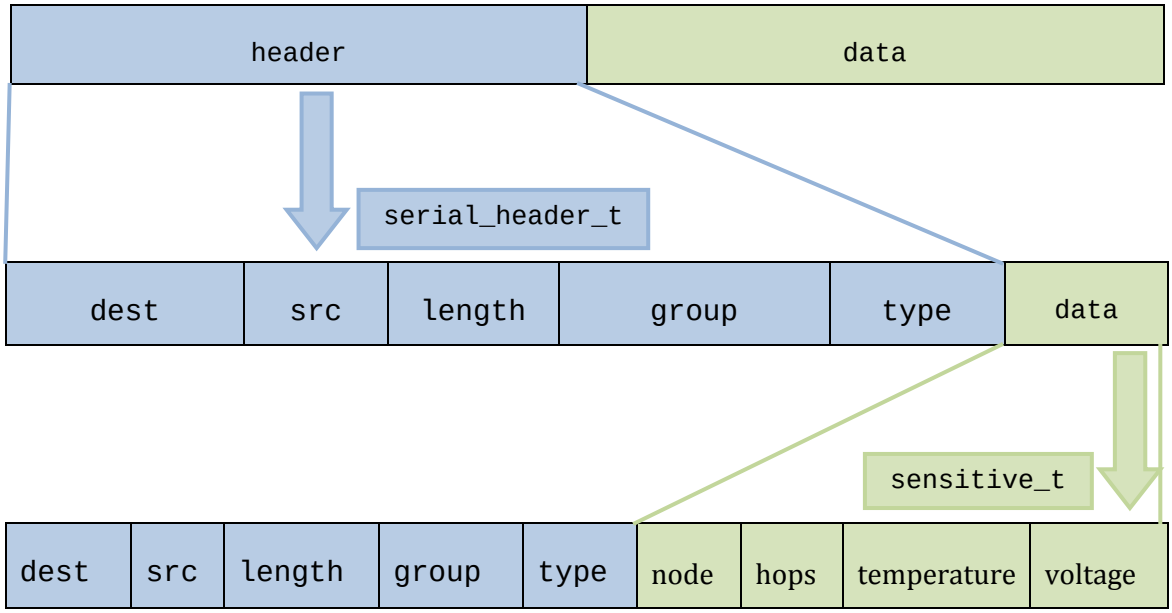


Figura 51: Estructura del paquete de datos serial

5.4 Diseño de la aplicación *SensitiveListen*

SensitiveListen es una aplicación totalmente distinta de las ya explicadas anteriormente (*SensitiveSampler* y *SensitiveBase*) ya que se ejecutará en un computador convencional a diferencia de las otras dos que se ejecutarán en sistemas embebidos. La utilización de las tres aplicaciones formará un único sistema con un objetivo común (obtener y almacenar la temperatura ambiente y el estado de las baterías). En este apartado se explica el diseño de *SensitiveListen.*

5.4.1 Funcionamiento

SensitiveListen es la aplicación encargada de leer todos los paquetes de datos que llegan por el puerto serial de la estación base, interpretar los datos obtenidos y almacenarlos en una base de datos. Para ello, el diseño de la aplicación ha partido de herramientas existentes en TinyOS para la captura de los paquetes entrantes vía serial. Estas herramientas se encuentran en el directorio `tinyos2.x/support/sdk/java/net/tinyos/tools`. La herramienta *Listen* servirá como base para el diseño de *SensitiveListen.*

La aplicación *SensitiveListen* tiene como objetivo realizar las siguientes funciones:

- Escuchar por un puerto (indicado por parámetro).
- Recibir correctamente todos los paquetes de datos provenientes del nodo raíz (gateway).
- Interpretar el paquete de datos recibido, obteniendo los valores y transformarlos en unidades del Sistema Internacional en cada caso.
- Con los datos obtenidos, insertarlos en la base de datos.

Como ya se ha mencionado anteriormente *SensitiveListen* es una aplicación que se ejecuta en un computador convencional, por tanto será implementada en un lenguaje de uso general (Java), ya que no tiene las restricciones de los sistemas embebidos (nodos sensores). Pero al igual que los casos anteriores la aplicación se puede entender muy bien con un diagrama de estados:

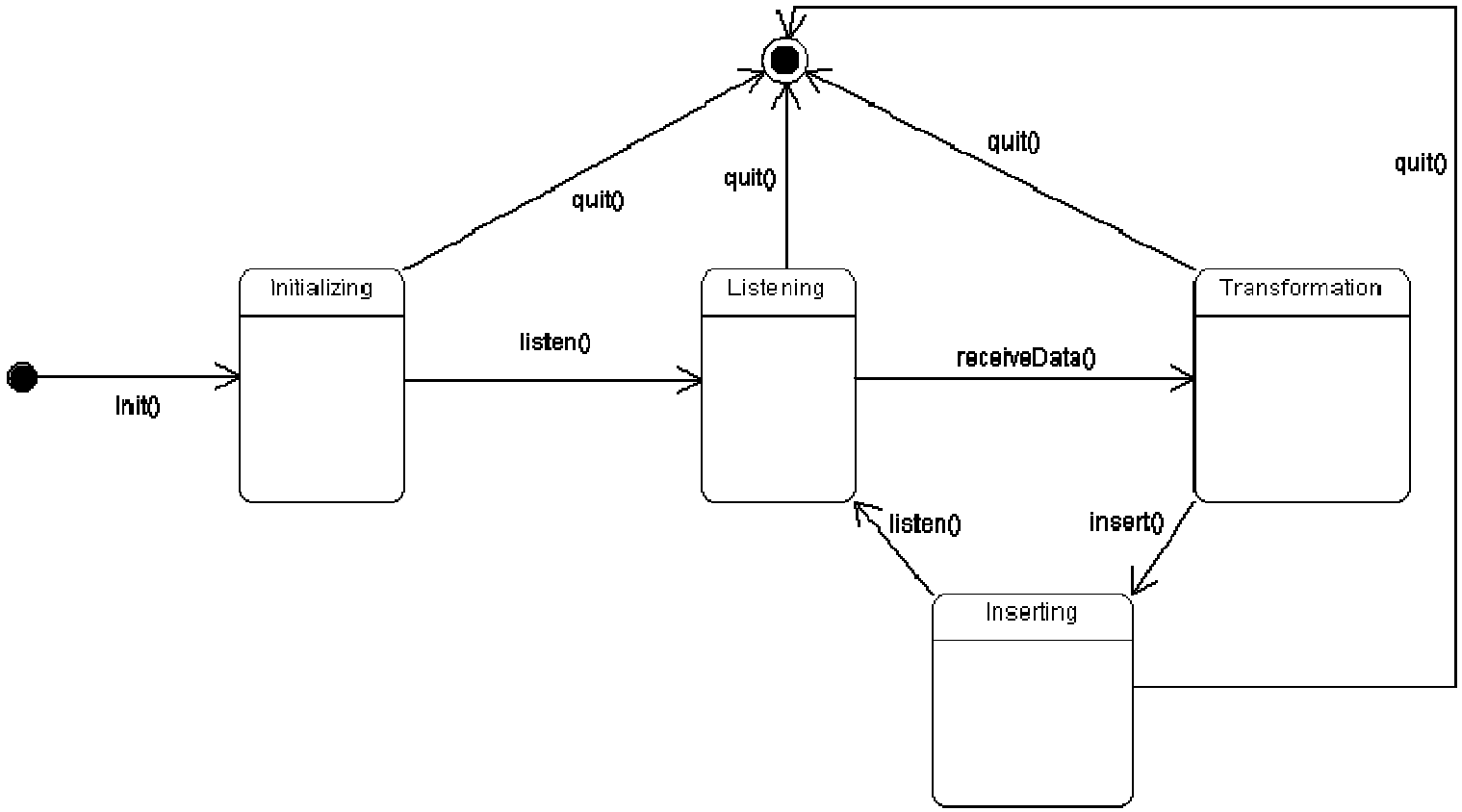


Figura 52: Diagrama de estados de la aplicación *SensitiveListen*

5.4.2 Estructura

Como puede observarse en la Figura 52 la aplicación consta de tres partes importantes, recepción de paquetes, transformación de datos e inserción en la base de datos.

5.4.2.1 Recepción de paquetes

La aplicación debe escuchar en un puerto determinado. Éste se indicará por parámetro. Por este puerto irán llegando los paquetes de datos y la aplicación debe leer todos y cada uno de ellos sin que haya pérdida de información. Para ello se utilizarán las clases Java que incluye TinyOS 2.x en su instalación:

- *PacketSource*: proporciona los mecanismos necesarios para recibir los paquetes de datos provenientes de la red de sensores. *PaketSource* implementa:
  - `readPacket()`: obtiene un paquete de datos sin formato.
  - `open(Messenger messages)`: abre la fuente de paquetes.
  - `close()`: cierra la fuente de paquetes.
- *BuildSource*: En esta clase se crea la fuente de paquetes especificado por cadenas de caracteres. *BuildSource* proporciona:
  - `makePacketSource(String name)`: crea una fuente de paquetes con el nombre especificado. Por defecto “sf@localhost:9002”.

5.4.2.2 Conversión de datos

Una vez obtenido un paquete de datos es preciso transformarlos en unidades del Sistema Internacional. Para ello se necesita conocer la estructura de la trama de datos recibido (véase 5.3.4 Formato paquete serial) y aplicar las formulas proporcionadas en la documentación de la placa sensora en cada caso. Para la conversión de la temperatura la fórmula es:

```
1/T(K) = a + b × ln(Rthr) + c × [ln(Rthr)]3
Donde:
      Rthr = R1(ADC_FS-ADC)/ADC
      a = 0.00130705
      b = 0.000214381
      c = 0.000000093
      R1 = 10 kΩ
      ADC_FS = 1023
      ADC = Medición del sensor de temperatura.
```

En el caso del voltaje:

```
Voltaje(V) = 1223 * 1024 / V * 1000
Donde:
      V = Medición de la CPU del voltaje.
```

5.4.2.3 Inserción en base de datos

Una vez obtenidos y transformados los datos del paquete recibido se procede a su inserción en la base de datos. Para ello, previamente ha de crearse una cadena de conexión con el servidor de la base de datos que se encontrará instalado en el mismo equipo. Una vez obtenida la cadena de conexión, por cada paquete recibido hay que crear una sentencia de inserción y lanzarla contra el servidor de base de datos. Se utilizarán las siguientes clases:

- `Connection`: crea una conexión (sesión) a una base de datos específica.
- `PreparedStatement`: representa una sentencia SQL precompilada.

En el siguiente apartado se describe la BBDD utilizada en este proyecto.

5.4.3 Modelo de datos

Se utilizará el servidor de base de datos MySQL 5.1.37. Se instala muy fácilmente y funciona a la perfección bajo el Sistema Operativo Ubuntu 9.10 (Linux 2.6.31-17-generic) que es el sistema operativo que gobierna la estación base. Además se instalarán dos herramientas para la gestión del servidor y para la gestión de la base de datos que se cree en él. La primera es *MySQL Administrator*, esta herramienta permite, entre otras muchas funciones, configurar el servidor (cambiando el puerto de acceso, directorios, etc.). También permite crear y ejecutar copias de seguridad (*backups*) de las bases de datos. La segunda

herramienta es *MySQL Query Browser,* es una utilidad para trabajar con la base de datos MySQL. Es un editor de sentencias SQL visual, que además incorpora herramientas para optimizar las consultas. Dispone también de un editor de tablas y registros, que permite crear nuevas tablas o cambiar las existentes y la posibilidad de cambiar los registros, es decir, los datos almacenados en las tablas.

Para el desarrollo del sistema se creará una base de datos (*Sensitive*) que consta de una sola tabla (*Samples*). Y en ella se almacenará la siguiente información:

- **Id**: identificador genérico del dato a insertar. Será campo clave de la tabla.
- **Date**: fecha y hora del momento de inserción en la base de datos. Tendrá el formato "YYYY-MM-DD hh:mm:ss" donde:
  - YYYY: año.
  - MM: mes.
  - DD: día.
  - hh: hora.
  - mm: minutos.
  - ss: segundos.
- **Node**: identificador del nodo al que pertenecen los datos.
- **Hops**: número de saltos realizados hasta llegar al nodo raíz.
- **Temperature**: valor de la temperatura obtenida del sensor (º Celsius).
- **Voltage**: valor del voltaje de las baterías (V).

Capítulo 6 Implementación de la WSN

En este capítulo se aborda la construcción del sistema para la WSN diseñada en el capítulo anterior, es decir, la codificación de las aplicaciones que conforman el sistema. Como se ha visto en puntos anteriores, el sistema, consta de tres aplicaciones. Por ello, la estructura de este capítulo se divide en tres apartados principales (uno por cada aplicación a implementar).

6.1 Implementación *SensitiveSampler*

Como ya se ha comentado en capítulos anteriores, *SensitiveSampler*, será la aplicación que resida en los nodos sensores y será la encargada de:

- Obtener datos de temperatura y voltaje.
- Enviar el paquete de datos a través de la radio.
- Interceptar para enrutar aquellos mensajes provenientes de otros nodos que lo precisen.

Para la implementación de *SensitiveSampler* será precisa la creación de cuatro ficheros:

- `Makefile`: este fichero contiene las reglas de compilación y es invocado cuando se compila la aplicación, ya sea para simular sobre el PC o para instalar en el mote.
- `Sensitive.h`: contiene la definición de la estructura de datos y de las constantes de las que hace uso la aplicación. Este fichero es compartido con la aplicación *SensitiveBase.*
- `SensitiveSamplerAppC.nc`: se trata del fichero de configuración y en él se definen los componentes de los que hace uso la aplicación y las relaciones entre ellos.
- `SensitiveSamplerC.nc`: a partir de la configuración establecida, este fichero contiene toda la lógica de la aplicación.

A continuación se mostrará la implementación de todas las funciones que ha de realizar la aplicación *SensitiveSampler*.

6.1.1 Definición del mensaje

Es necesario establecer una estructura con los datos que posteriormente se enviarán por radio. Estos datos se definen en el fichero `Sensitive.h` y tienen la siguiente forma:

```
typedef nx_struct sensitive{
      nx_uint16_t node;
      nx_uint16_t hops;
      nx_uint16_t temperature;
      nx_uint16_t voltage;
} sensitive_t;
```

El nombre de la estructura es `sensitive`. Para que la aplicación tenga acceso a la estructura y pueda utilizarla, hay que incluir la siguiente línea en el fichero de implementación del módulo (`SesintiveSamplerC.nc`):

```
#include “Sensitive.h”
```

La estructura contiene los siguientes campos:

- `node`: identificador del nodo al que pertenecen los datos.
- `hops`: número de saltos que realizará el paquete de datos hasta llegar al nodo raíz. Para ello, cada nodo que intercepte el paquete debe aumentar en una unidad este campo y reenviarlo a la red.
- `temperature`: contiene el valor de temperatura recogido a través del sensor.
- `voltage`: contiene el valor del nivel de voltaje de las baterías.

Además, en el fichero `Sensitive.h`, hay que establecer una constante para definir el tipo de mensaje de la aplicación y la parametrización de la interfaz de envío/recepción de mensajes. Tiene la siguiente forma:

```
AM_SENSITIVE = 0x94
```

6.1.2 Inicialización

Para inicializar la aplicación, se utiliza el componente `Main`. Este es un componente que representa el cuerpo `main` al estilo de un programa escrito en C y que necesita la interfaz `Boot` para su implementación.

Con la utilización de este componente el módulo `SensitiveSamplerC.nc` se comporta como una aplicación. Para ello, se ha de definir y configurar en el fichero de configuración `SensitiveSamplerAppC.nc`:

```
components MainC, SensitiveSamplerC;
```

Y su correspondiente *wiring*:

```
SensitiveSamplerC.Boot -> MainC;
```

Para utilizar la interfaz `Boot` se debe escribir la siguiente línea en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Boot;
```

Al utilizar la interfaz `Boot` se debe implementar el siguiente evento:

```
event void Boot.booted(){}
```

Este evento se lanza una vez que el componente `MainC` se ha iniciado correctamente. En `Boot.booted()` se deben iniciar aquellos componente que precisen también de inicialización y que para esta aplicación son: `ActiveMenssageC` y `CollectionC`. Para ello es preciso acceder a estos componentes a través de la interfaces `SplitControl` y `StdControl` respectivamente. Para ello, en `SensitiveSamplerAppC.nc` se escribe:

```
SensitiveSamplerC.RadioControl -> ActiveMessageC;
SensitiveSamplerC.RoutingControl -> CollectionC;
```

En el apartado `uses` del fichero `SensitiveSamplerC.nc` ha de indicarse las interfaces que se utilizan.

```
interface SplitControl as RadioControl;
interface StdControl as RoutingControl;
```

Con esto ya se pueden iniciar los componentes en el evento `Boot.booted()` de la siguiente forma:

```
event void Boot.booted(){
      if(call RadioControl.start() != SUCCESS){
            fatal_problem();
      }
      if(call RoutingControl.start() != SUCCESS){
            fatal_problem();
      }
}
```

Al utilizar la interfaz `SplitControl` para el componente `ActiveMessageC` ha de implementarse el evento `startDone()` que se lanza cuando el comando `start()` ha finalizado. El evento tiene la siguiente forma:

```
event void RadioControl.startDone(error_t error){
      if(error != SUCCESS){
            fatal_problem();
      }else{
            call Timer.startPeriodic(TIMER_INTERVAL);
            call LPL.setLocalSleepInterval(LPL_INTERVAL);
      }
}
```

El evento tiene como parámetro `error_t error` que indica si la inicialización del componente `ActiveMessageC` ha finalizado correctamente. Si es así, se inicia el *timer* y se establece el intervalo para *LowPowerListening* (véanse apartados 61.3.1 Implementación del Timer y 6.1.7 Ahorro de energía).

6.1.3 Obtención de datos

La aplicación ha de capturar los valores tanto del sensor de temperatura como el del nivel de las baterías en intervalos de tiempo configurables, por ejemplo 64 segundos. Por lo tanto, es necesario la implementación de un *timer* para que este señalice, a través de un evento, el momento de proceder a realizar las lecturas.

6.1.3.1 Implementación del Timer

Para la implementación del *Timer*, lo primero es definir la constante que marcará el intervalo de tiempo que debe transcurrir entre una lectura y la siguiente:

```
TIMER_INTERVAL = 64000U
```

La precisión del *timer* es de milisegundos, por lo tanto los 64 segundos hay que pasarlos a milisegundos (`64000`). Como en el caso de la constante `AM_SENSITIVE`, esta nueva constate se define el fichero `Sensitive.h`.

Como puede observarse la constante tiene un valor de `64000U`. La U, indica que la constante es sin signo (*unsigned*), se utiliza para valores mayores de `32768`. El motivo de su uso es evitar errores en tiempo de compilación.

Para la utilización de un *timer* hay que instanciar el componente en el fichero de configuración (`SensitiveSamplerAppC.nc`):

```
components new TimerMilliC();
```

Y realizar su correspondiente *wiring*:

```
SesitiveSamplerC.Timer -> TimerMilliC;
```

Tras definir el *timer* y su intervalo de tiempo, ya se puede implementar. Para ello, se debe escribir la siguiente línea en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Timer<TMilli>;
```

Con esto, se hace uso de la interfaz `Timer`, la cual obliga a implementar el evento `Timer.fired()`. Para iniciar el por primera vez el *timer* se realiza la invocación del comando:

```
call Timer.startPeriodic(TIMER_INTERVAL);
```

Con este comando se establece que el *timer* lance el evento `Timer.fired()` periódicamente cada `TIMER_INTERVAL` milisegundos. Esta llamada ha de realizarse justo después de que todos los componentes de la aplicación se hayan inicializado correctamente (véase apartado 6.1.2 Inicialización).

Como se ha mencionado anteriormente la interfaz `Timer` obliga a implementar el evento:

```
event void Timer.fired(){}
```

Dentro de este bloque se inicia un proceso de lecturas de los sensores: el de temperatura y el de voltaje. Este proceso se documenta a continuación.

6.1.3.2 Obtener muestra de temperatura

Antes de implementar la lectura del sensor de temperatura, es preciso configurar el archivo `PhotoTempDeviceC.nc`, ubicado en `/tos/sensorboards/mts300`, para obtener valores correctos de temperatura. Esto es debido a que la placa sensora utilizada en este proyecto es el modelo MTS310CB y el componente `TempC` viene configurado por defecto para la placa sensora MTS310CA. El cambio a realizar es el siguiente:

```
TempControl.Power -> MicaBusC.Int2;
```

Cambiarlo por:

```
TempControl.Power -> MicaBusC.PW0;
```

Una vez realizado el cambio ya es posible obtener el valor de la temperatura ambiente. Y para ello, es necesario instanciar el componente `TempC` en el fichero de configuración `SensitiveSamplerAppC.nc`:

```
components new TempC() as TempC;
```

Y realizar su correspondiente *wiring*:

```
SesitiveSamplerC.Temperature -> TempC;
```

Tras definir el `TempC` ya se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Read<uint16_t> as Temperature;
```

Con esto, se consigue usar la interfaz `Read`, que permite acceder al componente `TempC`. Esto obliga a implementar el evento `Temperature.readDone()` que ocurre cuando la lectura iniciada mediante `Temperature.read()` ha finalizado.

Para evaluar si la lectura de la temperatura se realiza con éxito se usa:

```
if(Temperature.read() != SUCCESS){
      report_problem();
}
```

Esta llamada se realiza dentro del evento `Timer.fired()` y se encapsula dentro de un bloque `if{}`, ya que TinyOS 2.x da la posibilidad de saber si la llamada a un comando de una interfaz se realiza correctamente o no.

Como se ha mencionado anteriormente la interfaz `Read` obliga a implementar el evento:

```
event void Temperature.readDone(error_t result, uint16_t data){}
```

En el evento ha de comprobarse que la lectura se ha realizado correctamente, los datos leídos se reciben por parámetro: `uint16_t data`. Tras esto, ha de almacenarse en la estructura de la aplicación el dato obtenido:

```
if(result != SUCCESS){
      data = 0xFFFF;
}
local.temperature = data;
```

Finalmente, la estructura de la aplicación contiene el nuevo dato de temperatura que posteriormente se enviará.

6.1.3.3 Obtener muestra del voltaje

Para obtener el valor del voltaje de las baterías es necesario instanciar el componente `VoltageC` en el fichero de configuración `SensitiveSamplerAppC.nc`:

```
components new VoltageC() as VoltageC;
```

Y realizar su correspondiente *wiring*:

```
SesitiveSamplerC.Voltage -> VoltageC;
```

Tras definir el `VoltageC` ya se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Read<uint16_t> as Voltage;
```

Como en el caso de la temperatura, se hace uso de la interfaz `Read`, que permite acceder al componente `VoltageC`. Obligando a implementar el evento `Voltage.readDone()` que ocurre cuando la llamada al comando `Voltage.read()` ha finalizado.

Análogamente, para evaluar si la lectura del voltaje se ha realizado con éxito:

```
if(Voltage.read() != SUCCESS){
      report_problem();
}
```

Como en el caso de la temperatura, esta llamada se realiza dentro del evento `Timer.fired()` y se encapsula dentro de un bloque `if` para saber si la llamada se realiza correctamente o no.

Como se ha mencionado anteriormente la interfaz `Read` obliga a implementar el evento:

```
event void Voltage.readDone(error_t result, uint16_t data){}
```

Se comprueba que la lectura se ha realizado correctamente y se guarda en la estructura de la aplicación:

```
if(result != SUCCESS){
      data = 0xFFFF;
}
local.voltage = data;
```

Tras esto, la estructura de la aplicación contiene el nuevo dato de voltaje que posteriormente se enviará.

Así, es como quedaría implementado el evento `Timer.fired()`.

```
event void Timer.fired(){
      if(Temperature.read() != SUCCESS){
            report_problem();
      }
      if(Voltage.read() != SUCCESS){
            report_problem();
      }
}
```

Se puede observar como las lecturas se realizan en paralelo, es decir, no se acaba de leer la temperatura del sensor para leer el voltaje (la lectura con `read()` no termina hasta que se lanza el evento `readDone()`). En el evento `readDone()`se inicia el envío del mensaje:

```
event void Temperature.readDone(error_t result, uint16_t data){
      if(result != SUCCESS){
            data = 0xFFFF;
      }
      local.temperature = data;
      sendSensitiveMsg();
}
```

Y en el caso del voltaje:

```
event void Voltage.readDone(error_t result, uint16_t data){
      if(result != SUCCESS){
            data = 0xFFFF;
      }
      local.voltage = data;
      sendSensitiveMsg();
}
```

Se realiza la llamada a la función `sendSensitiveMsg()` en dos ocasiones una por cada `readDone()`. Esto es así, porque al realizar las lecturas en paralelo no se sabe, a priori, cual acabará antes. Y para no enviar dos veces el mismo mensaje, se ha de controlar cuando las dos lecturas han terminado. Para ello, se utilizará dos variables booleanas que indicarán cuando las lecturas tanto de temperatura como de voltaje han finalizado y solo la segunda llamada a `sendSensitiveMsg()` sea la que envíe los datos. La solución es la siguiente:

```
bool temperatureReadDone = FALSE;
bool voltageReadDone = FALSE;
```

Se declaran las dos variables booleanas que indicarán cuando las lecturas han terminado correctamente, y se reinicializan cada vez que el *timer* lance el evento `fired()`:

```
event void Timer.fired(){
      temperatureReadDone = FALSE;
      voltageReadDone = FALSE;
      if(Temperature.read() != SUCCESS){
            report_problem();
      }
      if(Voltage.read() != SUCCESS){
            report_problem();
      }
}
```

Tras las llamadas a `read()`, en cada `readDone()`, se modificará cada variable para indicar que la lectura ha finalizado:

```
event void Temperature.readDone(error_t result, uint16_t data){
      temperatureReadDone = TRUE;
      if(result != SUCCESS){
            data = 0xFFFF;
      }
      local.temperature = data;
      sendSensitiveMsg();
}
```

Y en caso del voltaje:

```
event void Voltage.readDone(error_t result, uint16_t data){
      voltageReadDone = TRUE;
      if(result != SUCCESS){
            data = 0xFFFF;
      }
      local.voltage = data;
      sendSensitiveMsg();
}
```

Por tanto, antes de enviar el mensaje con los datos anteriormente leídos hay que preguntar si las dos lecturas han finalizado:

```
void sendSensitiveMsg(){
      if(temperatureReadDone && voltageReadDone){
                  …

      }
}
```

De esta manera se controla que el mensaje se envía con los dos datos leídos correctamente. El hecho de realizar las lecturas en paralelo se debe a que de esta manera se optimiza el uso del sensor y, por lo tanto, se ahorra en su consumo de energía.

6.1.4 Envío de mensajes

Para el envió de un mensaje a la red es necesaria la declaración de dos variables:

```
sensitive_t local;
message_t packet;
```

La variable `packet` es la estructura del mensaje que se enviará por radio y que en su *payload* (*array* de bytes reservados para datos) albergará la estructura `local` que es la que contendrá los datos de temperatura, voltaje, identificador del nodo y los saltos que realizará a través de la red. Tanto el identificador del nodo (*node*) como el número de saltos (*hops*) son valores que no variarán a lo largo de la ejecución. Por tanto se inicializan en el evento `Boot.booted()`, de la siguiente manera:

```
local.node = TOS_NODE_ID;
local.hops = 0;
```

El valor constante del sistema `TOS_NODE_ID` se define por parámetro en el momento de instalar la aplicación en el nodo sensor (véase apartado 8.2.2 Instalación de la aplicación en los motes).

Tras la lectura de la temperatura y el voltaje y almacenados en la estructura de la aplicación, se puede enviar la información obtenida. Para ello, es necesaria la instanciación del componente `CollectionSenderC` en el fichero `SensitiveSamplerAppC.nc`:

```
components new CollectionSender(AM_SENSITIVE);
```

Y realizar su correspondiente *wiring*:

```
SensitiveSamplerC.Send -> CollectionSenderC;
```

Tras su definición y configuración se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Send;
```

Como se hace uso de la interfaz `Send`, que permite acceder al componente `CollectionSenderC`, se puede realizar la llamada a `Send.send()`. Que tiene la siguiente forma:

```
call Send.send(&packet, sizeof local);
```

Donde el primer parámetro es el paquete de datos a enviar y el segundo es el tamaño que tiene la estructura (`sensitive_t`) con los datos que se han obtenido previamente. Antes de realizar la llamada a `send()` es necesario copiar los datos obtenidos de las lecturas previas dentro del paquete de datos a enviar, para ello:

```
memcpy(call Send.getPayload(&packet, sizeof local), &local, sizeof
local);
```

Con la función `memcpy` se copia el contenido de la estructura `local` dentro del mensaje en el espacio reservado a datos (*payload*).

La utilización de la interfaz `Send` obliga a implementar el siguiente evento:

```
event void Send.sendDone(message_t * msg, errot_t error){}
```

Donde el primer parámetro es el mensaje enviado y el segundo indica si el envió se ha realizado con éxito o no. Si el mensaje no se ha enviado correctamente, entonces se postea una tarea para que lo vuelva a reenviar y será TinyOS quien planifique cuando lo hace:

```
event void Send.sendDone(message_t * msg, errot_t error){
      if(error == SUCCES){
            reset_leds();
      }else{
            post sendSensitiveMsgTask();
      }
}
```

Como se ha mencionado anteriormente `sendSensitiveMsgTask()` es una tarea y se declara de la siguiente manera:

```
task void sendSensitiveMsgTask();
```

Y tiene la siguiente implementación:

```
task void sendSensitiveMsgTask(){
      sendSensitiveMsg();
}
```

Es decir, si el envío del mensaje ha fallado se postea una tarea para que TinyOS reintente el envío mas tarde. Por otro lado, es necesario añadir una variable booleana que actuará como semáforo. Esto es necesario porque es posible que se intente enviar un nuevo mensaje mientras ya se esté enviando otro anterior. Para ello:

```
bool sendBusy=FALSE;
```

Tras definir la variable, cuando se realiza la llamada a `Send.send()` en la función `sendSenstiveMsg()` y esta se realiza con éxito, se debe poner la variable a `TRUE.`

```
void sendSensitiveMsg(){
      if(temperatureReadDone && voltageReadDone){
            if(!sendBusy){
                  memcpy(call Send.getPayload(&packet, sizeof
                  local), &local sizeof local);
                  if(call Send.send()&packet,sizeof
                  local)==SUCCESS){
                        sendBusy=TRUE;
            }else{
                  post sendSensitiveMsgTask();
            }
      }
}
```

Se puede observar lo primero que se hace en la función `sendSenstiveMsg()` es comprobar que la variable `sendBusy` no esté a `TRUE`. Cuando el envío se ha realizado con éxito la variable `sendBusy` se pone a `FALSE` de la siguiente forma:

```
event void Send.sendDone(message_t * msg, errot_t error){
      sendBusy = FALSE;
      if(error == SUCCES){
            reset_leds();
      }else{
            post sendSensitiveMsgTask();
      }
}
```

6.1.5 Recepción de mensajes

Para la recepción de mensajes es necesaria la utilización del componente `CollectionC`, que como en los casos anteriores se define:

```
components CollectionC as Collector;
```

Y su correspondiente *wiring*:

```
SensitiveSamplerC.Receive -> Collector.Receive[AM_SENSITIVE];
```

El componente `CollectionC` provee varias interfaces, una de ellas es `Receive` que en su instanciación necesita del parámetro `AM_SENSITIVE` para recibir los mensajes correctamente. Tras su definición y configuración se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Receive;
```

La utilización de esta interfaz requiere la implementación del siguiente evento:

```
event message_t * Receive.receive(message_t *msg, void *payload,
uint8_t len){}
```

En *SensitiveSampler* no es necesario realizar ninguna acción con los mensajes entrantes, ya que no es el encargado de reenviarlos a la estación base. Pero, por motivos de la implementación del componente `CollectionC` es necesario tener configurada la interfaz `Receive`. Como se puede observar el evento `Receive.receive()` es necesario retornar el mensaje recibido por parámetro, por lo cual su implementación queda:

```
event message_t * Receive.receive(message_t *msg, void *payload,
uint8_t len){
      return msg;
}
```

6.1.6 Reenvío de mensajes

Para los mensajes que llegan al nodo y necesitan ser reenviados a niveles superiores en la jerarquía del árbol es preciso la utilización del componente `CollectionC` a través de la interfaz `Intercept`. A continuación se muestra su *wiring*:

```
SensitiveSamplerC.Intercept -> Collector.Intercept[AM_SENSITIVE];
```

Como en el caso de `Receive`, es necesario parametrizar la interfaz `Intercept` con el tipo de mensaje `AM_SENSITIVE`. Tras su definición y configuración se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Intercept;
```

Esta interfaz obliga a implementar el siguiente evento:

```
event bool Intercept.forward(message_t *msg, void *payload, uint8_t
len){}
```

El evento `fordward()` recibe por parámetro el mensaje recibido, el puntero que apunta al principio del *payload* y el tamaño en bytes del *payload*. Debe retornar una variable booleana para indicar si el mensaje debe ser reenviado o no. Su implementación es la siguiente:

```
event bool Intercept.forward(message_t *msg, void *payload, uint8_t
len){
      sensitive_t *s = (sensitive_t *) payload;
      s -> hops++;
      return TRUE;
}
```

Como se puede observar el parámetro que apunta al *payload* es un puntero sin tipo. Por lo que, lo primero es castearlo a un puntero del tipo `sensitive_t`. Tras esta operación es posible modificar el *payload* y aumentar en una unidad el número de saltos que realiza el paquete de datos a través de la red. Finalmente se devuelve siempre `TRUE` para hacer saber a las capas inferiores que el mensaje debe ser reenviado a la red.

6.1.7 Ahorro de energía

En este apartado se explica cómo usar el protocolo *LowPowerListening* en la aplicación. Para ello, es necesaria la utilización del componente `CC2420ActiveMensageC` a través de la interfaz `LowPowerListening`. Su declaración es la siguiente (`SensitiveSamplerAppC.nc`):

```
components CC2420ActiveMessageC as LPLRadio;
```

Y su correspondiente *wiring*:

```
SensitiveSamplerC.LPL -> LPLRadio;
```

Una vez declarado y configurado el componente, se debe especificar la interfaz en la sección `uses` de `SensitiveSamplerC.nc` que se emplea para acceder al componente:

```
interface LowPowerListening as LPL;
```

Tras esto, ya se puede usar *LowPowerListening* en la aplicación. Lo primero es establecer el intervalo de tiempo en que la radio debe estar dormida antes del envío de un paquete. Para calcular el intervalo se debe tener en cuenta el intervalo del *timer* y el tamaño del mensaje para calcular el tiempo suficiente que debe despertase antes la radio para escuchar los preámbulos de mensaje. Para ello se utiliza una fórmula obtenida de la documentación de Crossbow Technology Inc:

```
time_rx_msg = 1000 * msg_size_bytes * 8 / Baud_rate

Donde:
      msg_size_bytes: Tamaño del mensaje a enviar.
      Baud_rate: Tasa transmisión por segundo en un red, por defecto
      (38400)
```

Se obtiene un valor de `time_rx_msg=6,3 milisegundos`. Para obtener los milisegundos que se precisan para recibir todos los mensajes necesarios:

```
time_rx_all_msg = 2 * 5 milisegundos * time_rx_msg
```

Se obtiene un valor de `63 milisegundos`, que redondeando se dejará en `70 milisegundos`. El valor para el intervalo de LPL será:

```
LPL_INTERVAL = TIMER_INTERVAL - time_rx_all_msg
```

Tras calcular el intervalo de para *LowPowerListening* hay que definir la constante en `sensitive.h`:

```
LPL_INTERVAL=63930U
```

Ahora en necesario establecer el ciclo de trabajo para *LowPowerListening*. Se realiza en el mismo momento que al establece el *timer* en el evento `RadioControl.startDone()`:

```
event void RadioControl.startDone(error_t error){
      if(error != SUCCESS){
            fatal_problem();
      }else{
            call Timer.startPeriodic(TIMER_INTERVAL);
            call LPL.setLocalSleepInterval(LPL_INTERVAL);
      }
}
```

Con esto, se establece que del ciclo de 64 segundos entre una lectura y otra la radio debe despertase 70 milisegundos antes para recibir paquetes de otros nodos. Es decir, más del 99% del tiempo la radio se encuentra en estados de consumo energético ultra bajo.

Por otro lado, hay que configurar el mensaje saliente de manera que se indique el intervalo de *LowPowerListening* que tiene el nodo emisor. Para ello se realiza la llamada al siguiente comando:

```
call LPL.setRxSleepInterval(&packet, LPL_INTERVAL);
```

Este comando recibe por parámetro el paquete a enviar y el intervalo *LowPowerListening* con el que está configurado el nodo. Por último, solo queda hacer la llamada justo antes de enviar el mensaje:

```
void sendSensitiveMsg(){
      if(temperatureReadDone && voltageReadDone){
            if(!sendBusy){
                  memcpy(call Send.getPayload(&packet, sizeof
                  local), &local sizeof local);
                  call LPL.setRxSleepInterval(&packet,
                  LPL_INTERVAL);
                  if(call Send.send()&packet,sizeof
                  local)==SUCCESS){
                        sendBusy=TRUE;
            }else{
                  post sendSensitiveMsgTask();
            }
      }
}
```

6.1.8 Configuración del fichero de compilación

Para construir de forma correcta el fichero ejecutable, se han de usar una serie de parámetros en un fichero denominado *Makefile*, que es invocado cuando se compila la aplicación. El contenido del fichero es el siguiente:

```
SENSORBOARD=mts300
COMPONENT=SensitiveSamplerAppC
CFLAGS += -I..
CFLAGS += -I$(TOSDIR)/lib/net/
CFLAGS += -I$(TOSDIR)/lib/net/ctp
CFLAGS += -I$(TOSDIR)/lib/net/4bitle
CFLAGS += -DLOW_POWER_LISTENING
include $(MAKERULES)
```

Donde:

- El parámetro `SENSORBOARD` incluye el tipo de placa sensora que se utiliza en el mote.
- El parámetro `COMPONENT` debe incluir el nombre del componente de configuración de la aplicación.
- El parámetro `CFLAGS` define las rutas de las cabeceras, componentes e interfaces que se utilizan que no son propios del sistema.
- `include $(MAKERULES)`: fichero que contiene el conjunto de todas las reglas de compilación.

6.2 Implementación *SensitiveBase*

SensitiveBase será la aplicación que resida en el nodo gateway y será la encargada de:

- Recibir los paquetes de datos provenientes de la red de sensores.
- Enviar todos los paquetes de datos, recibidos de la red de sensores, vía puerto serial a la estación base.

Para la implementación de *SensitiveBase* será precisa la creación de cuatro ficheros:

- `Makefile`: este fichero contiene las reglas de compilación. Es invocado cuando se compila la aplicación, ya sea para simular sobre el PC o para instalar en el mote.

- `Sensitive.h`: contiene la definición de la estructura de datos y de las constantes de la que hace uso la aplicación. Este fichero es compartido con la aplicación *SensitiveSampler.*
- `SensitiveBaseAppC.nc`: se trata del fichero de configuración y en él se definen los componentes de los que hace uso la aplicación y las relaciones entre ellos.
- `SensitiveBaseC.nc`: a partir de la configuración establecida, este fichero contiene toda la lógica de la aplicación.

A continuación se mostrará la implementación de todas las funciones que ha de realizar la aplicación *SensitiveBase*.

6.2.1 Inicialización

Como en el caso de *SensitiveSampler,* para inicializar la aplicación se utiliza el componente `Main` y que necesita de la interfaz `Boot` para su implementación.

Con la utilización de este componente, el módulo `SensitiveBaseC.nc` se comporta como una aplicación. Para ello, se ha de definir y configurar en el fichero de configuración `SensitiveBaseAppC.nc`:

```
components MainC, SensitiveBaseC;
```

Y su correspondiente *wiring*:

```
SensitiveBaseC.Boot -> MainC;
```

Para utilizar la interfaz `Boot` se debe escribir la siguiente línea en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Boot;
```

Al utilizar la interfaz `Boot` se debe implementar el siguiente evento:

```
event void Boot.booted(){}
```

En `Boot.booted()` se deben iniciar aquellos componentes que precisen también de inicialización y que para esta aplicación son: `SerialActiveMenssageC`,

ActiveMenssageC y CollectionC. En este caso se utilizarán tres interfaces para acceder a los tres componentes. En SensitiveSamplerAppC.nc se escribe:

```
Components CollectionC as Collector, ActiveMessageC,
SerialActiveMessage;

SensitiveSamplerC.RadioControl -> ActiveMessageC;
SensitiveSamplerC.SerialControl -> SerialActiveMessageC;
SensitiveSamplerC.RoutingControl -> CollectionC;
```

En el apartado uses del fichero SensitiveSamplerC.nc ha de indicarse las interfaces que se utilizan:

```
interface SplitControl as RadioControl;
interface SplitControl as SerialControl;
interface StdControl as RoutingControl;
```

Con esto ya se pueden iniciar los componentes en el evento Boot.booted() de la siguiente forma:

```
event void Boot.booted(){
      if(call RadioControl.start() != SUCCESS){
            fatal_problem();
      }
      if(call RoutingControl.start() != SUCCESS){
            fatal_problem();
      }
}
```

Al utilizar la interfaz SplitControl para el componente ActiveMessageC ha de implementarse el evento startDone() que se lanza cuando el comando start() ha finalizado. El evento tiene la siguiente forma:

```
event void RadioControl.startDone(error_t error){
      if(error != SUCCESS){
            fatal_problem();
      }
      if(call SerialControl.start() != SUCCESS){
            fatal_problem();
      }
}
```

El evento tiene como parámetro error del tipo error_t que indica si la inicialización del componente ActiveMessageC ha finalizado correctamente. Como SerialActiveMessageC utiliza la interfaz SplitControl se ha de implementar el evento startDone():

```
event void SerialControl.startDone(error_t error){
      if(error != SUCCESS){
            fatal_problem();
      }
      call RootControl.setRoot()
}
```

Se puede observar cómo se realiza la llamada al comando `RootControl.setRoot()` este comando establece al nodo como el nodo raíz del protocolo que implementa el componente `CollectionC`. En el siguiente apartado se explica con más detalle.

6.2.2 Establecerse como nodo raíz

Para establecer como nodo raíz o *root* ha de utilizarse el componente `CollectionC` a través de la interfaz `RootControl`. Para ello, precisa el siguiente *wiring*:

```
SensitiveBaseC.RootControl -> Collector;
```

Tras su configuración se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveBaseC.nc`) la interfaz a utilizar:

```
interface RootControl;
```

La interfaz `RootControl` proporciona los mecanismos necesarios para que el nodo funcione como nodo raíz. Tan solo hay que utilizar el comando `setRoot()`después de la inicialización de la aplicación y sus componentes que lo precisen.

```
event void SerialControl.startDone(error_t error){
      if(error != SUCCESS){
            fatal_problem();
      }
      call RootControl.setRoot()
}
```

6.2.2 Recepción de mensajes por radio

Para la recepción de mensajes del resto de nodos de la red es necesaria la utilización del componente `CollectionC`. Que como en los casos anteriores se define:

```
components CollectionC as Collector;
```

Y su correspondiente *wiring*:

```
SensitiveSamplerC.Receive -> Collector.Receive[AM_SENSITIVE];
```

El componente `CollectionC` provee la interfaz `Receive` que en su instanciación necesita del parámetro `AM_SENSITIVE` para recibir los mensajes de los nodos que deberán tener la misma configuración. Tras su definición y configuración se puede implementar. Para ello, se debe indicar en la sección `uses` en el fichero del módulo (`SensitiveSamplerC.nc`):

```
interface Receive;
```

La utilización de la interfaz `Receive` requiere la implementación del siguiente evento:

```
event message_t * Receive.receive(message_t *msg, void *payload,
uint8_t len){}
```

En *SensitiveSampler* no era necesario realizar ninguna acción con los mensajes entrantes, pero al tratarse del nodo raíz *SensitiveBase* debe realizar las siguientes funciones:

- Recibir correctamente un mensaje proveniente de la red de sensores.
- Si no se está enviando un mensaje anterior, enviar vía serial el mensaje entrante.
- En caso contrario almacenar el mensaje entrante para que posteriormente se enviado vía serial a través de una tarea.

La implementación de estas acciones se describe en el siguiente apartado.

6.2.3 Envío de mensajes por puerto serie

Cuando se recibe un mensaje por radio se lanza el evento `receive()` de la interfaz `Receive`. En este evento se implementará el grueso de la funcionalidad de la aplicación. Para ello, será necesaria la utilización de las siguientes variables:

```
sensitive_t *in = (sensitive_t*) payload;
sensitive_t *out;
```

El puntero `*in` apunta al *payload* del paquete de datos entrantes. Y el puntero `*out` apuntará al *payload* del paquete serial saliente. Como en el caso de *SensitiveSampler* será preciso el uso de una variable semáforo para evitar situaciones en la que se intente enviar un mensaje mientras el envío por serial ya esté en uso:

```
bool uartbusy = FALSE;
```

En el evento `receive()` según el estado de la variable `uartbusy` se realizará unas u otras acciones. Si `uartbusy` es `FALSE` (el envío por serial no está ocupado):

- Se comprueba que el tamaño del *payload* del mensaje entrante.
  - Si el tamaño es erróneo no se realiza ninguna acción.
  - Si el tamaño es correcto se copia el contenido del *payload* del mensaje entrante al mensaje serial saliente. Y se postea el envío del mensaje vía serial.

Si `uartbusy` es `TRUE` (el envío por serial está ocupado):

- Se encola el mensaje en la buffer de mensajes que tienen que enviarse vía serial.

El evento `Receive.receive()` queda implementado de la siguiente manera:

```
event message_t* Receive.receive(message_t* msg,void *payload,
uint8_t len){
      sensitive_t* in = (sensitive_t*)payload;
      sensitive_t* out;
      if (uartbusy == FALSE) {
            out = (sensitive_t*)call SerialSend.getPayload(&uartbuf,
            sizeof(sensitive_t));
            if (len != sizeof(sensitive_t) || out == NULL) {
                  return msg;
            }
            else {
                  memcpy(out, in, sizeof(sensitive_t));
            }
            post uartSendTask();
      } else {
            message_t *newmsg = call UARTMessagePool.get();
            if (newmsg == NULL) {
                  report_problem();
                  return msg;
            }
            out = (sensitive_t*)call SerialSend.getPayload(newmsg,
            sizeof(sensitive_t));
            if (out == NULL) {
                  return msg;
            }
            memcpy(out, in, sizeof(sensitive_t));
            if (call UARTQueue.enqueue(newmsg) != SUCCESS) {
                  call UARTMessagePool.put(newmsg);
                  report_problem();
                  return msg;
            }
      }
      return msg;
}
```

Como puede observarse el evento utiliza dos componentes distintos para almacenar aquellos mensajes entrantes que no pueden ser enviados a la estación base en el momento. Estos componentes son `PoolC` y `QueueC`. Se trata de componentes virtualizados, por lo tanto, deben ser instanciados:

```
components new PoolC(message_t, 10) as UARTMessagePoolP,
new QueueC(message_t*, 10) as UARTQueueP;
```

Esos componentes reciben dos parámetros en su instanciación. El primero, es el tipo de datos que contendrán, y el segundo, el número de elementos que almacenará. A continuación se muestra su correspondiente *wiring*:

```
SensitiveBaseC.UARTMessagePool -> UARTMessagePoolP;
SensitiveBaseC.UARTQueue -> UARTQueueP;
```

`UARTMessagePool` será el componente encargado de contener los mensajes entrantes que no puedan ser enviados al momento, es decir, es un contenedor de mensajes. Mientras que `UARTQueue` contendrá los punteros a dichos mensajes, es decir, es una cola que almacena las direcciones de memoria donde empieza cada mensaje que debe ser enviado vía serial.

Para utilizar estos componentes, en el fichero de módulo `SensitiveBaseC.nc` debe indicarse las interfaces que acceden a ellos:

```
interface Queue<message_t *> as UARTQueue;
interface Pool<message_t> as UARTMessagePool;
```

Las interfaces utilizadas son `Queue<typedef_t>` y `Pool<typedef_t>`. Como puede observarse son interfaces parametrizadas por el tipo de datos que contendrán.

La interfaz `Pool<typedef_t>` permite alojar elementos (`get`) y desalojarlos (`put`). Por otro lado, la interfaz `Queue<typedef_t>` permite encolar (`enqueue`) elementos al final de la cola y desencolar elementos (`dequeue`) que obtiene y quita de la cabeza de la cola un elemento.

Cuando se quiere enviar un paquete de datos vía serial se postea la tarea `uartSendTask()`, que tiene la siguiente forma:

```
task void uartSendTask() {
      if (call SerialSend.send(0xffff, &uartbuf,
      sizeof(sensitive_t)) == SUCCESS) {
            uartbusy = TRUE;
      } else {
            report_problem();
      }
}
```

Para el envío de paquetes serial es necesario la utilización del componente `SerialAMSenderC(AM_TYPE)`. Para ello:

```
components new SerialAMSenderC(AM_SENSITIVE);
```

Y su correspondiente *wiring*:

```
SensitiveBaseC.SerialSend -> SerialAMSenderC.AMSend;
```

En el fichero del módulo (`SensitiveBaseC.nc`) en la sección `uses` debe indicarse la interfaz que accede al componente:

```
interface AMSend as SerialSend;
```

La interfaz `SerialSend` permite la utilización del comando `send()` para enviar mensajes vía serial. Como siempre, la interfaz obliga a implementar eventos, en este caso:

```
event void sendDone(message_t *msg, error_t error){}
```

El evento ocurre cuando el envío de un mensaje vía serial ha finalizado, el evento indica por parámetro si el comando `send()` se ha realizado con éxito o no. `sendDone` tiene la siguiente forma:

```
event void sendDone(message_t *msg, error_t error){
      uartbusy = FALSE;
      if (call UARTQueue.empty() == FALSE) {
            message_t *queuemsg = call UARTQueue.dequeue();
            if (queuemsg == NULL) {
                  report_problem();
                  return;
            }
            memcpy(&uartbuf, queuemsg, sizeof(message_t));
            if (call UARTMessagePool.put(queuemsg) != SUCCESS) {
                  report_problem();
                  return;
            }
            post uartSendTask();
      }
      report_bridged();
}
```

Tras el envío de un mensaje, se comprueba si hay más mensajes por enviar, es decir, si hay almacenado algún menaje pendiente para enviar. Si es así, se obtiene el mensaje a enviar y se postea el envío con la llamada a la tarea `uartSendTask()`.

6.2.4 Configuración fichero de compilación

Como en caso de *SensitiveSampler*, *SensitiveBase* debe utilizar un fichero (Makefile) para construir el fichero ejecutable, que es invocado cuando se compila la aplicación. El contenido del fichero es el siguiente:

```
COMPONENT=SensitiveBaseAppC
CFLAGS += -I..
CFLAGS += -I$(TOSDIR)/lib/net/
CFLAGS += -I$(TOSDIR)/lib/net/ctp
CFLAGS += -I$(TOSDIR)/lib/net/4bitle
include $(MAKERULES)
```

Donde:

- El parámetro `COMPONENT` debe incluir el nombre del componente de configuración de la aplicación.
- El parámetro `CFLAGS` define las rutas de las cabeceras, componentes e interfaces que se utilizan que no propios del son de sistema.
- `include $(MAKERULES)`: fichero que contiene el conjunto de todas las reglas de compilación.

6.3 Implementación *SensitiveListen*

La aplicación que se encargará de escuchar los mensajes enviados por serial de *SensitiveBase* y que reside en la estación base es *SensitiveListen,* y ha de cumplir con las siguientes funciones:

- Escuchar por un puerto y recibir los paquetes de datos provenientes del nodo raíz (gateway).
- Interpretar y transformar los paquetes de datos recibidos.
- Visualizar por pantalla los datos en crudo y los datos transformados.
- Insertar en la base datos la información recibida.

Para la implementación de *SensitiveListen* será precisa la creación de un sólo fichero:

- `SensitiveListen.java`: en él se encuentra toda la lógica del programa. A partir de este archivo se genera la clase Java (.class).

A continuación se mostrará la implementación de todas las funciones que ha de realizar la aplicación *SensitiveListen*.

6.3.1 Obtención de parámetros

Cuando se invoque la aplicación ésta ha de recibir un parámetro indicando el puerto por el que debe escuchar. Al tratarse de una aplicación Java, la clase debe implementar la función:

```
public static void main(String args[]) throws Exception {}
```

Al estilo de las aplicaciones en C, la función `main` tiene como parámetro un *array* de cadenas de caracteres. En este array se almacenan los parámetros que se indican en el momento de invocar la aplicación. En el caso de *SensitiveListen* tan sólo recibirá uno (puerto de escucha para recibir mensajes). Para ello es preciso:

```
public static void main(String args[]) throws Exception {
      String source = null;
      if (args.length == 2 && args[0].equals("-comm")) {
            source = args[1];
      }
      else if (args.length > 0) {
            System.err.println("usage: java net.tinyos.tools.Listen
            [-comm PACKETSOURCE]");
            System.err.println(" (default packet source from MOTECOM
            environment variable)");
            System.exit(2);
      }
      …
      …
}
```

Puede observarse como la variable `source` contiene la cadena de caracteres que indica el puerto a escuchar, si en la invocación a la aplicación no se indica ningún puerto por defecto escuchará en `sf@localhost:9002`. En el caso de que el formato o el número de parámetros sea erróneo la aplicación mostrará el error y saldrá a través de la llamada a `System.exit(2)`.

Una vez se conoce el puerto, se puede inicializar la clase que permitirá escuchar los paquetes entrantes. Esta clase pertenece a la API que ofrece TinyOS en su instalación. Para ello, se debe importar:

```
import java.tinyos.packet.*;
```

Crear la variable del objeto en cuestión:

```
PacketSource reader;
```

E inicializar `reader` con el puerto a escuchar a través de la clase estática `BuildSource`:

```
if (source == null) {
      reader = BuildSource.makePacketSource();
}
else {
      reader = BuildSource.makePacketSource(source);
}
if (reader == null) {
      System.err.println("Invalid packet source (check your MOTECOM
      environment variable)");
      System.exit(2);
}
```

Si el parámetro `source` es erróneo la aplicación mostrará el error y saldrá a través de la llamada a `System.exit(2)`.

6.3.2 Conexión a base de datos

Antes de recibir paquetes, es necesario establecer la cadena de conexión con la base de datos. Esta conexión se realizará a través de JDBC y para ello es necesario importar las clases que lo implementan:

```
import java.sql.*;
```

Ahora es posible instanciar e inicializar las clases que obtendrán la conexión (`Connection con`) y la sentencia SQL precompilada (`PreparedStatement pst`) para la inserción de datos:

```
Connection con = null;
PreparedStatement pst=null;
try {
      Class.forName("com.mysql.jdbc.Driver");
      con =
      DriverManager.getConnection("jdbc:mysql://localhost:3306/sensi
      tive?","root","root");
      pst=con.prepareStatement("insert into samples
      (date,node,hops,temperature,voltage) values(NOW(),?,?,?,?)");
      System.out.println("Connection established: " + con);
} catch (Exception e) {
      System.out.println("Error getting connection to " +
      "localhost:3306/sensitive.");
      e.printStackTrace();

}
```

`Class.forName()` es un método estático que genera una clase para el controlador JDBC, es decir un objeto Driver específico para MySQL que define cómo se ejecutan las instrucciones para esa base de datos en particular.

Con la llamada a la función `DriverManager.getConnection()` se especifica la conexión al servidor de la base de datos que en este caso se encuentra en el mismo computador (`localhost`). Finalmente se configura la sentencia SQL que insertará los datos en la tabla *samples*.

6.3.3 Leer mensajes de la red de sensores

Una vez inicializados y configurados todos los elementos que intervienen en la aplicación, se inicia el proceso de recepción de mensajes, transformación e inserción en la base de datos.

6.3.3.1 Obtener mensaje

Para iniciar el proceso de obtención del mensaje es necesario abrir la fuente de paquetes, para ello:

```
reader.open(PrintStreamMessenger.err);
```

Para leer los paquetes de datos entrantes:

```
data = reader.readPacket();
```

`readPacket()` retorna un *array* de bytes sin formato. Es decir, que para obtener los datos es preciso conocer la estructura del mensaje serial (véase apartado 5.3.4 Formato del mensaje serial) y realizar las conversiones oportunas. En el siguiente apartado se explica el proceso de conversión de datos.

6.3.3.2 Transformar datos

La trama de datos recibida vía serial es:

Tabla 5: Estructura de la trama de datos serial

00	dest (2byte)	src (2byte)	length (1byte)	group (1byte)	type (1byte)	node (2byte)	hops (2byte)	temperature (2byte)	voltage (2byte)

Los datos que se insertarán en la base de datos son los 4 últimos de la trama. Es necesario conocer el tamaño y el orden que ocupan dentro de la trama para poder transformarlos en los tipos de datos correctos para su inserción en la base de datos, para ello:

```
int node;
int hops;
int temperature;
int voltage;
```

Se declaran las variables que almacenarán los datos. Es preciso recordar que el sensor no da valores en unidades del Sistema Internacional, por ello, se precisa la declaración de otras dos variables:

```
float temp_converted;
float volt_converted;
```

La conversión del tipo de datos byte a `integer` se realiza con la función `getIntegerElement()`, de la siguiente forma:

```
//Obtiene valores del payload
node = getIntegerElement(data,8);
hops = getIntegerElement(data,10);
temperature = getIntegerElement(data,12);
voltage = getIntegerElement(data,14);
```

La función `getIntegerElement()` recibe dos parámetros. El primero, la trama de datos en bruto, y el segundo, la posición de inicio del dato a convertir en el *array*. La función tiene la siguiente forma:

```
private static int getIntegerElement (byte[] arr, int offset) {
      int high = arr[offset] & 0xFF;
      int low = arr[offset+1] & 0xFF;
      return (int)( high << 8 | low );
}
```

Tras las operaciones de conversión la función devuelve un *integer*. Como se ha mencionado anteriormente la temperatura y el voltaje precisan pasarse a unidades del Sistema Internacional. Para ello:

```
temp_converted = (float)((1/(0.001307050 + 0.000214381 *
                  Math.log( (10000 * (1023-temperature))
                  /temperature ) + 0.000000093 *
                  Math.pow(Math.log( (10000 * (1023-
                  temperature))/temperature ),3))) - 273.15);
```

Y para la conversión del voltaje:

```
volt_converted = (float) (1223 * 1024) / (voltage * 1000);
```

Para más información sobre las fórmulas de conversión de temperatura y voltaje véase el apartado 5.4.2.2 Conversión de datos.

6.3.3.3 Visualización de los datos en pantalla

Otra de las funciones de la aplicación es mostrar por pantalla la trama de datos recibida y los datos transformados ante de su inserción en la base de datos:

```
System.out.println("============================================");
System.out.println("Payload:");
System.out.println("--------------------------------------------");
Dump.printPacket(System.out, data);
System.out.println();
System.out.println("--------------------------------------------");
System.out.println("Conversion:");
System.out.println("--------------------------------------------");
System.out.println("Node: " + node);
System.out.println("Hops: " + hops);
System.out.println("Temperature: " + temp_converted);
System.out.println("Voltage: " + volt_converted);
System.out.flush();
```

6.3.3.4 Inserción en la base de datos

Por último, queda la inserción de los datos obtenidos en la base de datos. Para ello, es preciso establecer los parámetros de la sentencia SQL de inserción:

```
//Prepara insert en BBDD y ejecuta el comando.
pst.setLong(1,node);//Node_id
pst.setLong(2,hops);//Hops
pst.setFloat(3,temp_converted);//Temperature
pst.setFloat(4,volt_converted);//Voltage
```

Una vez establecidos los parámetros para la inserción, queda la ejecución de la sentencia *insert*:

```
pst.executeUpdate();
```

Así quedan almacenados los datos obtenidos de la trama de datos. Para las posteriores inserciones hay que limpiar los parámetros, para ello se utiliza:

```
pst.clearParameters();
```

Este proceso debe repetirse cada vez que se reciba un mensaje, es decir, es necesario el empleo de un bucle infinito con las todas las funciones descritas anteriormente (obtener mensaje, transformar datos, visualizar en pantalla e insertar en la base de datos) de la siguiente manera:

```
for (;;) {
      //Obtiene la trama de datos del paquete
      data = reader.readPacket();
      //Obtiene los valores del payload de la trama de datos
      node = getIntegerElement(data,8);
      hops = getIntegerElement(data,10);
      temperature = getIntegerElement(data,12);
      voltage = getIntegerElement(data,14);
      //Conversion datos a unidades del SI
      temp_converted = (float)((1/(0.001307050 + 0.000214381 *
      Math.log( (10000 * (1023-temperature))/temperature ) +
      0.000000093 * Math.pow(Math.log( (10000 * (1023-
      temperature))/temperature ),3))) - 273.15);
      volt_converted = (float) (1223 * 1024) / (voltage * 1000);
      //Prepara insert en BBDD y ejecuta el comando.
      pst.setLong(1,node);//Node_id
      pst.setLong(2,hops);//Hops
      pst.setFloat(3,temp_converted);//Temperature
      pst.setFloat(4,volt_converted);//Voltage
      pst.executeUpdate(); //Insert
      pst.clearParameters();
      //Muestra salida en pantalla
      System.out.println("=======================================");
      System.out.println("Payload:");
      System.out.println("---------------------------------------");
      Dump.printPacket(System.out, data);
      System.out.println();
      System.out.println("---------------------------------------");
      System.out.println("Conversion:");
      System.out.println("---------------------------------------");
      System.out.println("Node: " + node);
      System.out.println("Hops: " + hops);
      System.out.println("Temperature: " + temp_converted);
      System.out.println("Voltage: " + volt_converted);
      System.out.flush();
}
```

Capítulo 7 Modelo de consumo de nodo MicaZ

En este capítulo se explicará el modelo utilizado para estimar el consumo de energía de un nodo MicaZ así como el cálculo del tiempo de vida dentro de la red de sensores. Como se ha visto a lo largo del documento la energía es un recurso muy limitado para un nodo. Por ello es muy deseable contar con un modelo que muestre los distintos consumos de energía según la funcionalidad de la aplicación instalada en la red.

7.1 Introducción

Los motes MicaZ disponen de un microcontrolador que proporciona distintos modos de funcionamiento que permiten el ahorro de energía (38).

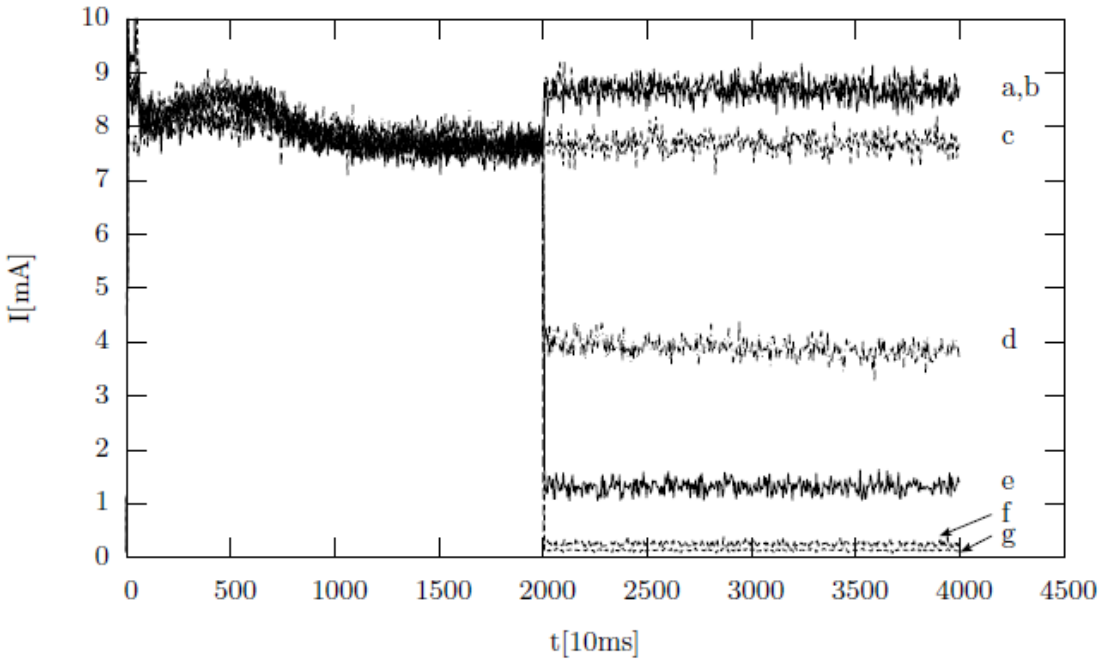


Figura 53: Modos de ahorro de energía del microprocesador Atmega 128L

La Figura 53 muestra los diferentes modos de funcionamiento, donde:

- a: busy (mul).
- b: busy (jmp).
- c: NOPLoop.

- d: idle.
- e: ADC.
- f: extended standby.
- g: save

Los primeros tres (a, b y c) se refieren a las mediciones de los modos de funcionamiento normal. Dos de ellos, `Busy(mul)` y `Busy(jmp)`, representan los ciclos de servicio y tienen un gran consumo, mientras que el tercero `NOPLoop` es un bucle de inactividad y tiene un menor consumo. El resto de modos (d, e, f y g) son los de mayor ahorro de energía. El modo de `Idle` tiene el menor ahorro, pero a cambio, tiene la ventaja de que casi todas las partes del microcontrolador todavía funcionan. Los modos `ADC`, `Extended Standby`, `Save` son los de mayor ahorro de energía, ya que deshabilitan varias partes del microcontrolador. En esta medición, el modo `Standby` y el modo de `Power Down` no han sido considerados. Estos dos modos detienen el reloj principal, por lo que no les es posible despertar por un tiempo determinado, sólo una interrupción externa puede despertar al dispositivo.

El modelo que se presenta en este proyecto está basado en AEON (Accurate Prediction of Power Consumption) (39) y adaptado para la plataforma MicaZ. Además, el simulador Avrora soporta este modelo proporcionando una herramienta de análisis de energía tras la simulación de una aplicación. AEON está basado en la ejecución real de la aplicación junto con el sistema operativo y la medición de la intensidad de corriente que precisa cada uno de los dispositivos que componen un nodo (CPU, radio, LEDs, sensor, memoria y flash). De esta manera se consigue con precisión datos sobre el consumo de un nodo y así determinar el tiempo de vida de éste.

7.2 Cálculo del consumo de energía de un nodo MicaZ

Para calcular la intensidad de corriente total consumida por un nodo deberían tenerse en cuenta todas las corrientes consumidas por cada uno de los componentes (dispositivos) que forman un mote. Para ello es preciso conocer:

- Componentes que se utilizan en la aplicación.

- Conocer los consumos (intensidad de corriente consumida) de cada estado de cada componente del nodo sensor.
- En cada dispositivo, calcular porcentualmente el tiempo que permanece en cada estado.
- Calcular, para cada estado, el consumo de cada componente y obtener el consumo total del nodo.

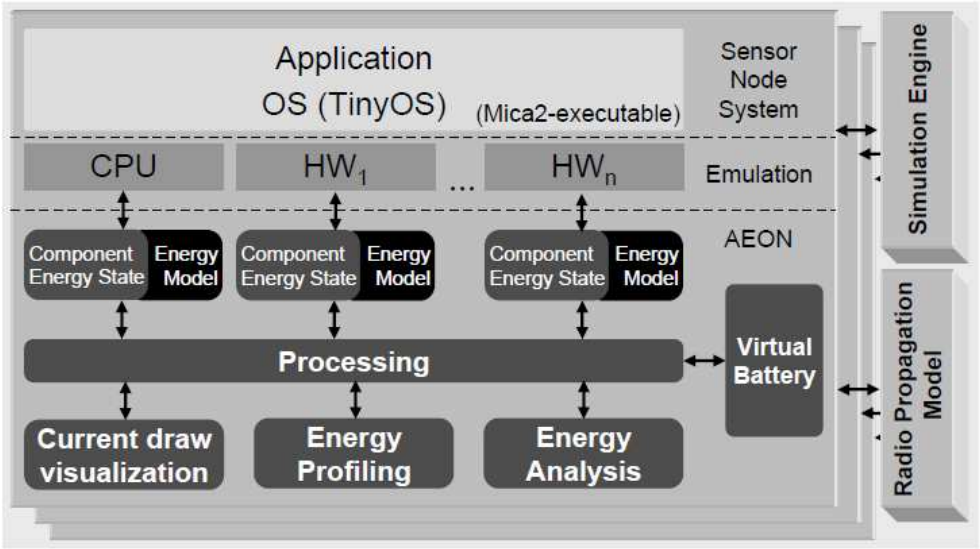


Figura 54: Diagrama de bloques de la arquitectura de AEON

De esta manera, la ecuación que permite conocer el consumo de un mote sería:

$$\text{Consumo nodo} = \sum \text{Consumo componente}$$

Donde,

$$\sum \text{Consumo componente} \begin{cases} \text{Consumo CPU} \\ + \\ \text{Consumo Radio} \\ + \\ \text{Consumo LEDs} \\ + \\ \text{Consumo Sensor} \\ + \\ \text{Consumo Flash} \end{cases}$$

7.2.1 Componentes usados

Durante el diseño de la aplicación se ha de determinar que componentes se utilizarán para que la aplicación cumpla las necesidades establecidas. Por ejemplo, la aplicación puede no necesitar almacenar datos en memoria flash ya que los datos obtenidos del sensor se envíen inmediatamente por radio. También es posible que la aplicación una vez testeada no necesite del uso de los LEDs para indicar el proceso en el que se encuentre la aplicación. Por ello, cada componente que no sea utilizado no entrará dentro del cómputo del consumo del nodo.

7.2.2 Consumos medidos de cada componente

Para poder determinar el consumo de un nodo durante la ejecución de una aplicación, primero hay que conocer la intensidad de corriente eléctrica que consumen los distintos dispositivos en sus distintos estados. Por ejemplo, la radio de un nodo sensor puede encontrarse enviando o recibiendo datos, en cada uno de estos estados la radio precisa de distintas intensidades de corriente para funcionar. Al igual que la radio, la CPU tiene distintos modos de funcionamiento (activa, en espera, durmiendo, apagada, etc.) y en cada uno consume una intensidad de corriente distinta. A mayor intensidad mayor será el consumo del nodo, por tanto, menor tiempo de vida del nodo en la red. A continuación se muestra una tabla con las intensidades consumidas por los dispositivos en sus diferentes estados de funcionamiento.

Tabla 6: Mediciones del consumo de los componentes de un nodo MicaZ

Dispositivo	Estado	Corriente consumida (1h)
CPU (Atm128)	Active	7,5 mA
	Idle	3,3 mA
	ADC Noise Reduction	988 µA
	Power Down	116 µA
	Power Save	124 µA
	Reserved 1	0 mA
	Reserved 2	0 mA
	Standby	236 µA
	Extended Standby	243 µA
Radio (CC2420)	Power Off	0,02 µA
	Power Down	20 µA
	Idle	426 µA

	Receive (Rx)	18,8 mA
	Transmit (Tx: 0dBm)	17,4 mA
LED (Cada uno)	On	2,2 mA
	Off	0 mA
Sensor (MTS310CB)	On	700 µA
	Off	0 mA
Memoria Flash	Standby	2 µA
	Read	4 mA
	Write	15 mA
	Load	2 µA

Es importante comentar que esta información pertenece a los datos recopilados en el modelo de AEON, implementados en el simulador Avrora. Las mediciones están realizadas mediante un osciloscopio y según la especificación de AEON tiene un error máximo del 5%.

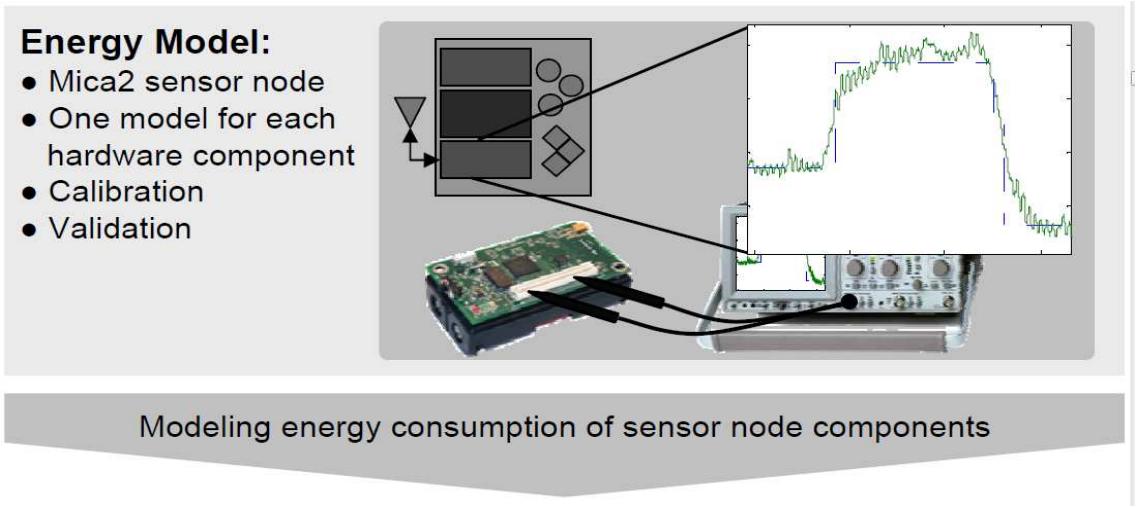


Figura 55: Modelo de consumo de energía del nodo sensor.

En la Tabla 6 se puede observar los consumos de los distintos componentes de un nodo MicaZ. El mayor consumo ocurre cuando la radio del nodo se encuentra escuchando en el canal para recibir mensajes de otros nodos (*listening*). Por ello es fundamental reducir los tiempos de recepción de mensajes para aumentar el tiempo de vida del nodo en la red. La CPU en estado activo también tiene un consumo de corriente elevado, por lo que será preciso que la aplicación instalada junto con el sistema operativo lleve a la CPU a estados mínimos de consumo el mayor tiempo posible. Los LEDs también son dispositivos que consumen una cantidad importante de energía por lo que su uso deberá estar restringido. Lo

mismo ocurre con la memoria flash. Dependiendo de las necesidades de la aplicación será necesario su uso o no, teniendo en cuenta el coste energético que tiene acceder a dicha memoria. Por tanto, el diseño de la aplicación instalada en los motes determinará el consumo y el tiempo de vida en la red, para obtener un diseño que cumpla con las limitaciones energéticas de un nodo sensor véase 3.2.1.4 Programando en NesC.

7.2.3 Cálculo porcentual de tiempos de activación de cada estado

Llegados a este punto es necesario realizar una simulación de la ejecución de la aplicación en un nodo sensor. Para ello es necesario utilizar el simulador Avrora (véase Anexo A Avrora). Para ello se realizarán simulaciones de la siguiente forma:

- Tiempo de simulación 1 hora (3600 segundos).
- Aplicación para gateway (nodo con identificador 0): *SensitiveBase.*
- Aplicación para nodo sensor (nodo con identificador 1): *SensitiveSampler.*

Una vez realizada la simulación, nos centraremos en los datos obtenidos del nodo sensor, que es el que se encuentra bajo la mayores restricciones de energía. Avrora, tras la simulación, muestra los ciclos de reloj totales de la simulación y los ciclos que cada componente en cada estado. A través de estos datos podemos determinar porcentualmente el tiempo en el que se encuentra cada componente en cada estado. A continuación se muestra un ejemplo del resultado de una simulación con Avrora:

```
=={ Energy consumption results for node 1 }==================================
Node lifetime: 26542080000 cycles,  3600.0 seconds

CPU: 6.635349830108519 Joule
  Active: 0.2852008044678141 Joule, 92630803 cycles
  Idle: 5.204689166747151 Joule, 3825873866 cycles
  ADC Noise Reduction: 5.116333393554687E-4 Joule, 1272147 cycles
  Power Down: 0.0 Joule, 0 cycles
  Power Save: 1.132162902920288 Joule, 22493157237 cycles
  RESERVED 1: 0.0 Joule, 0 cycles
  RESERVED 2: 0.0 Joule, 0 cycles
  Standby: 0.0 Joule, 0 cycles
  Extended Standby: 0.012785322633911134 Joule, 129145947 cycles

Yellow: 5.371093750000001E-9 Joule
  off: 0.0 Joule, 26542079994 cycles
  on: 5.371093750000001E-9 Joule, 6 cycles

Green: 2.3417991684570314 Joule
  off: 0.0 Joule, 23926077438 cycles
  on: 2.3417991684570314 Joule, 2616002562 cycles

Red: 5.371093750000001E-9 Joule
  off: 0.0 Joule, 26542079994 cycles
  on: 5.371093750000001E-9 Joule, 6 cycles

Radio: 29.502872632899297 Joule
  Power Off:          : 1.8437281791178385E-4 Joule, 22655731865 cycles
  Power Down:         : 2.0333513183593753E-4 Joule, 24985821 cycles
  Idle:           : 4.592302880859375E-4 Joule, 2649306 cycles
  Receive (Rx):       : 29.386353927083338 Joule, 3841484224 cycles
  Transmit (Tx):      31:   : 0.11567176757812501 Joule, 17228784 cycles

SensorBoard: 7.56 Joule
  on:  : 7.56 Joule, 26542080000 cycles

flash: 0.021599999999999998 Joule
  standby: 0.021599999999999998 Joule, 26542080000 cycles
  read: 0.0 Joule, 0 cycles
  write: 0.0 Joule, 0 cycles
  load: 0.0 Joule, 0 cycles
```

El cálculo del tiempo estimado de vida del nodo se realiza de la siguiente manera:

% Tiempo estado = Ciclos (Componente: Estado) * 100 / Ciclos totales

El tiempo de simulación es de 1 hora para facilitar cálculos posteriores con los datos de consumo de intensidad de corriente eléctrica.

7.2.4 Cálculo de consumo parcial y total del nodo

Conociendo el consumo de intensidad de la corriente eléctrica y el porcentaje de tiempo en el cual está activo un estado del componente (radio transmitiendo, CPU en *standby,* etc.) se puede calcular el consumo de una aplicación durante una hora. Para ello simplemente se multiplica el consumo de

corriente de cada componente en cada estado por el porcentaje en el que ha estado activo dicho estado del componente.

Consumo componente (A) = Consumo (Amperios) * % Componente: Estado

Una vez obtenido el consumo real de cada componente se puede obtener el consumo total del nodo sensor en 1 hora. Para ello:

Consumo nodo (Amperios) = ∑ Consumo componente (Amperios)

7.2.5 Duración de las baterías

Llegados a esto punto se puede estimar la duración de las baterías, ya que normalmente su capacidad viene indicada en mAh (miliamperios hora). Por ejemplo, las baterías (Tipo AA) recargables Panasonic modelo HHR-3MRE tienen una capacidad de 2100 mAh, es decir, 2.1 Ah. La fórmula para obtener las horas de funcionamiento de la aplicación instalada en el nodo sensor es:

Horas de funcionamiento (HF) = Capacidad pilas (Ah) / Consumo nodo (A)

7.3 Consideraciones

Con este modelo de consumo de energía se obtiene una estimación precisa sobre el consumo que realiza la aplicación y el tiempo de vida del nodo sensor. Por otro lado, Avrora muestra tras la simulación el consumo de energía (trabajo realizado) de cada componente en Julios, mientras que el modelo presentado en este capítulo lo hace en Amperios para facilitar los cálculos, ya que la capacidad de las baterías siempre viene indicada en mAh (miliamperios hora). Por otro lado, las simulaciones han sido realizadas con un solo nodo sensor y no se han tenido en cuenta el consumo de energía que llevaría a cabo la radio por el enrutamiento de los mensajes de nodos vecinos.

Capítulo 8 Evaluación

En este capítulo se presentan las pruebas realizadas y se comprueba el correcto funcionamiento de la aplicación. Las pruebas se pueden realizar en un entorno de simulación y en entorno real. En cada uno de ellos se prueban los diferentes escenarios propuestos para la evaluación.

Se evaluará el comportamiento de dos protocolos distintos de enrutamiento, el ya presentado *Collection Tree Protocol* (*CTP*) y el protocolo *Link Quality Indicator* (*LQI*), que tiene las mismas funcionalidades que CTP pero es una implementación específica para la radio CC2420 y más ligera que CTP. Por otro lado se evaluará la aplicación utilizando políticas de ahorro de energía (*Low Power Listening*) y sin ellas.

Para cada uno de estos escenarios se evaluará el comportamiento de la aplicación utilizando distintos intervalos de monitorización, a través de una serie de potencias de dos (2, 4, 8, 16, 32 y 64 segundos). Para entender los resultados de estas comprobaciones y obtener conclusiones sobre su comportamiento, se elaborarán una serie de gráficas para mostrar los resultados obtenidos. Además, para cada escenario se estimará la duración de las baterías en el nodo sensor con el modelo propuesto en el capítulo anterior.

8.1 Simulaciones con *Avrora*

Para el entrono de simulación se utilizará Avrora. Con este simulador se podrá comprobar el funcionamiento de las aplicaciones *SensitiveSampler* y *SensitiveBase*. Aunque el estudio se centrará en el consumo del nodo sensor ya que el nodo gateway se encuentra conectado vía USB a la estación base y no tiene las limitaciones energéticas del nodo sensor . Por otro lado, para comprobar el correcto funcionamiento de la aplicación *SensitiveListen* tendrá que realizarse su ejecución en un entorno real.

Las pruebas con Avrora tendrán la siguiente estructura:

Figura 56: Esquema de la red WSN en la simulación

Se simulará la ejecución de un nodo sensor enviando paquetes de datos al nodo gateway. Cada uno ejecutará la aplicación correspondiente, como indica en Figura 56.

La ejecución de las simulaciones tiene que desarrollarse en el mismo directorio donde se encuentren los ficheros ejecutables de cada aplicación, debido a que Avrora utiliza los ficheros binarios para realizar las simulaciones. Los parámetros para configurar la simulación son los siguientes:

- Plataforma (*platform*): indica la plataforma en la que están construidos los nodos sensores. Se usará el valor *micaz*.
- Simulación (*simulation*): tipo de simulación a realizar. Se usará el valor *sensor-network* para simular una red de sensores.
- Número de nodos (*nodecount*): indica el número de nodos que ejecutan las aplicaciones. Se usará el valor 1,1 para indicar un nodo por cada aplicación.
- Segundos (*seconds*): tiempo de la simulación. Se usará el valor 3600 para realizar 1 hora de simulación.
- Monitor (*monitors*): indica las diferentes variables a monitorizar durante la simulación. Se usará *energy*, el cual muestra un análisis sobre el consumo de cada nodo, y los ciclos de reloj que ha utilizado cada componente del nodo.

De esta forma el comando utilizado para la simulación con Avrora es:

```
java avrora.Main  –platform=micaz
                  –simulation=sensor-network
                  –nodecount=1,1
                  –seconds=3600
                  –monitors=packet,energy
                  SensitiveBase/SensitiveBase.elf
                  SensitiveSampler/SensitiveSampler.elf
```

Donde los ficheros objeto `SensitiveSampler.elf` y `SensitiveBase.elf` se encuentran en subdirectorios dentro del directorio que contiene las aplicaciones. Para más información sobre las posibilidades que ofrece el simulador, véase Anexo A Avrora.

Las simulaciones se realizarán en tres implementaciones distintas de *SensitiveSampler.* De esta manera se puede analizar el impacto que tiene el uso de las distintas políticas de ahorro de energía y protocolos de enrutamiento sobre el consumo de un nodo. Las tres implementaciones de *SensitiveSampler* sometidas a estudio son:

- *SensitiveSampler* usando como protocolo de enrutamiento *Collection Tree Protocol*, sin implementación de *Low Power Listening.*
- *SensitiveSampler* utilizando *Link Quality Indicator*, sin implementación de *Low Power Listening.*
- *SensitiveSampler* utilizando *Collection Tree Protocol* e implementando *Low Power Listening*.

Debe comentarse que la implementación del protocolo de enrutamiento *Link Quality Indicator* no permite la utilización de políticas de ahorro de energía como *Low Power Listening.* Por ello, no podrán realizarse las simulaciones en este caso.

Para cada caso se analizarán distintos de muestreo que oscilarán entre 2 segundos (Modelo 1) y 64 segundos (Modelo 6) para obtener mediciones de la energía.

Por otro lado, es importante señalar que Avrora no implementa en su análisis energético el componente de la placa sensora. Es decir, los datos que muestra sobre el consumo de la placa sensora corresponden al consumo de la placa si estuviese siempre activa (el peor de los casos). Se trata pues, de una de las funcionalidades que Avrora tiene pendientes en implementaciones de próximas versiones.

8.1.1 Escenario 1: Sensitive con Collection Tree Protocol (CTP)

Como se ha mencionado antes, se simulará la aplicación por cada intervalo de monitorización, empezando por 2 segundos hasta 64, siguiendo una escala de potencias de 2. En este caso la aplicación *SensitiveSampler* no implementa *LowPowerListening.* A continuación se muestran los resultados de los consumos energéticos para cada caso.

8.1.1.1 Modelo 1: 2 segundos

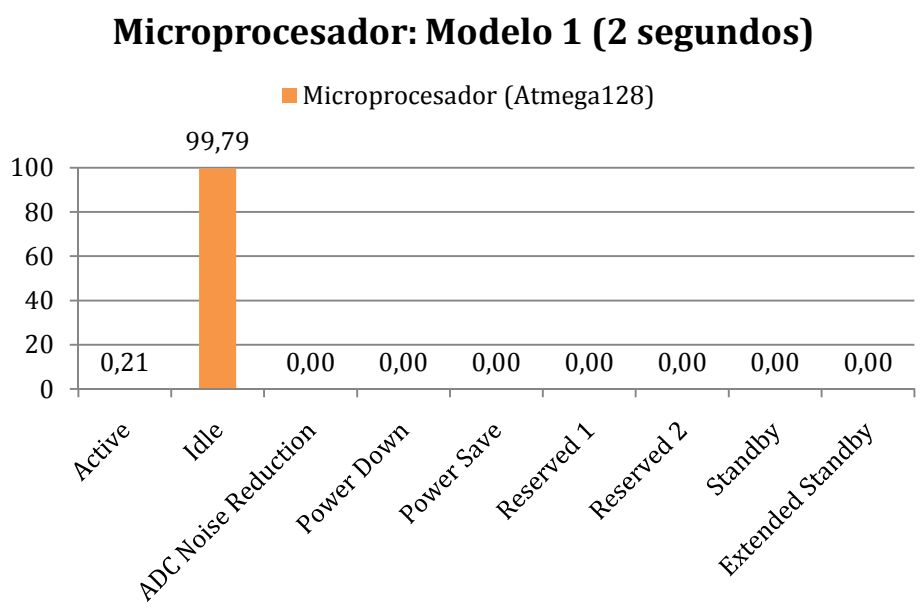
A continuación se muestra el resultado sobre el análisis de energía que *Avrora* proporciona tras finalizar la simulación. Se mostrarán los datos referentes al microprocesador y la radio, ya que estos son los dos componentes que producen casi la totalidad del consumo de energía en el nodo. A partir de estos datos es posible calcular los porcentajes de activación de cada componente y el consumo generado en amperios:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles,3600.0 seconds

CPU: 36.204115064677495 Joule
   Active: 0.1728460178754476 Joule, 56138921 cycles
   Idle: 36.031269046802045 Joule, 26485941079 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.9624078874629 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.84660314697268 Joule, 26516798505 cycles
   Transmit (Tx): 0.11580431945800783 Joule, 17248527 cycles
```

En la Figura 57 puede observarse que el microprocesador nunca entra en estados de mínimo consumo energético (estado `Power Save`). Esto se debe a que la radio, al estar en modo escucha, no permite que el microprocesador baje a dichos estados.



57: Porcentajes de activación de la CPU (Escenario 1: Modelo 1)

La siguiente gráfica (Figura 58) muestra como la radio se encuentra la prácticamente la totalidad del tiempo en estado de escucha (`Receive`). En el caso de la transmisión de paquete, la radio apenas está un 0,07% del tiempo enviando datos. Por otro lado, la radio prácticamente nunca se apaga, tan solo 0,03% del tiempo se encuentra en estado `Off`.

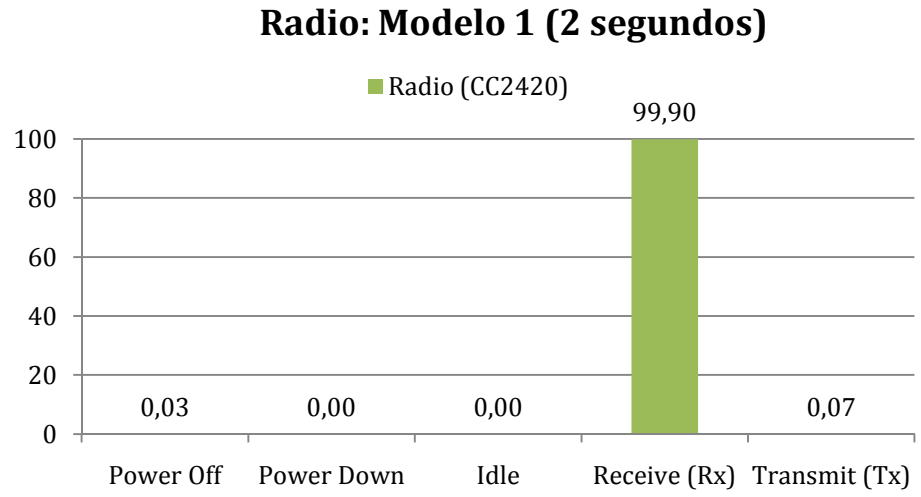


Figura 58: Porcentajes de activación de la radio (Escenario 1: Modelo 1)

En la Figura 59 se pueden observar el consumo en amperios de cada componente del nodo en una hora. La radio es, con diferencia, el componente que provoca un mayor consumo con 18.8 miliamperios. Esto se debe a que en este

escenario no se usa *Low Power Listening* para tener la radio escuchando el mínimo tiempo posible. El consumo del microprocesador también es elevado, con un gasto de 3.3 miliamperios, debido a que la radio se encuentra estado (`Receive`).

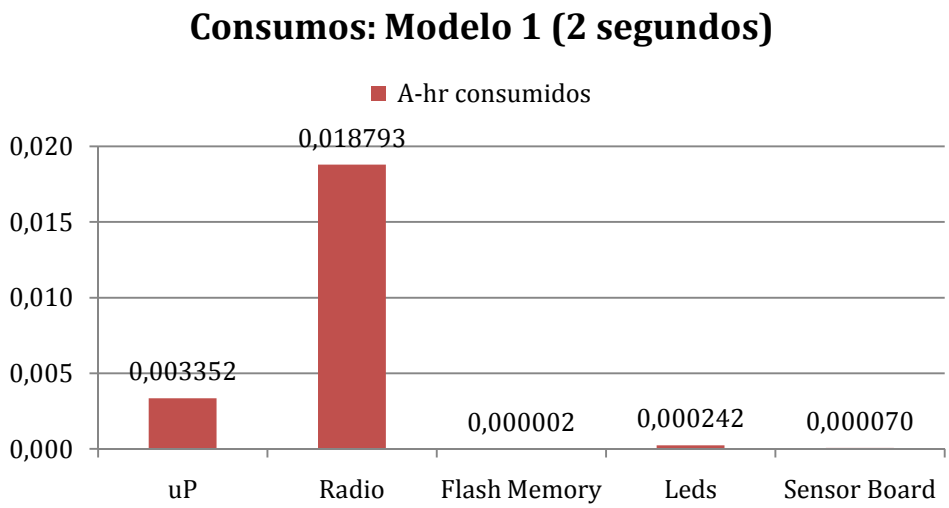


Figura 59: Consumos de los componentes (Escenario 1: Modelo 1)

8.1.1.2 Modelo 2: 4 segundos

El resultado del análisis energético tras simulación que muestra *Avrora* para el modelo 2 es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.168494629648436 Joule
   Active: 0.1090279694466146 Joule, 35411360 cycles
   Idle: 36.05946666020182 Joule, 26506668640 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97040908826853 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.91200426660157 Joule, 26525347962 cycles
   Transmit (Tx): 0.05840440063476563 Joule, 8699070 cycles
```

Los resultados que muestra *Avrora* en el modelo 2 son muy similares a los anteriores (modelo 1). Aunque se haya doblado el intervalo de monitorización, éste no tiene influencia en los porcentajes de tiempo en los que los componentes

están en estados de consumo elevados. La radio sigue escuchando en el medio el tiempo que transcurre entre una lectura y la siguiente, y por tanto, el microprocesador se encuentra en espera sin poder bajar a estados de mínimo consumo.

En la Figura 60 puede observarse que en el microprocesador no se producen cambios en los porcentajes de tiempo de activación de cada estado, a pesar de haber doblado el intervalo de monitorización.

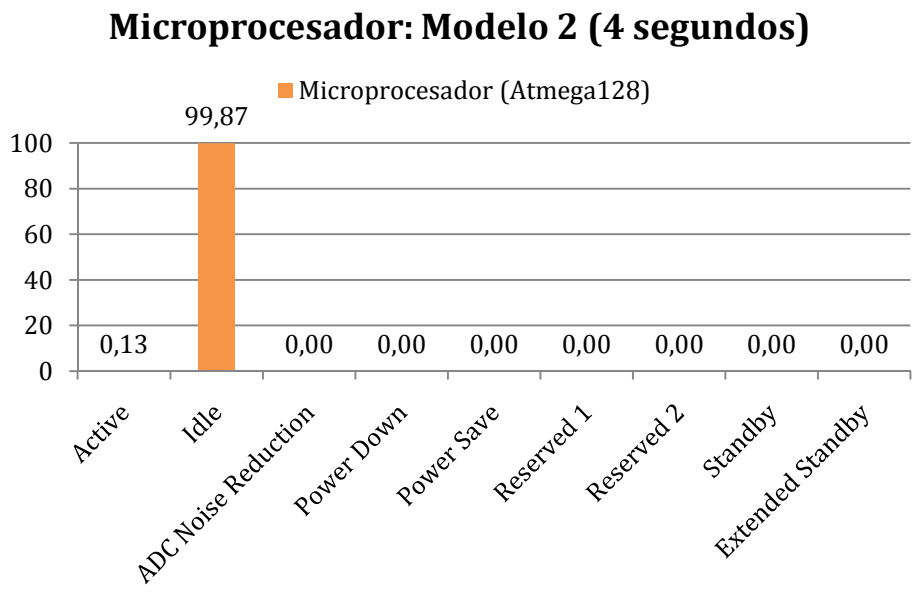


Figura 60: Porcentajes de activación de la CPU (Escenario 1: Modelo 2)

En el caso de la radio, el tiempo dedicado al envío de mensajes se ha reducido a la mitad, debido a que el tiempo de monitorización es el doble y se envían la mitad de paquetes de datos. Por otro lado, al reducirse el envió de mensajes aumenta el tiempo dedicado a la recepción de los mismos como puede observase en la Figura 61:

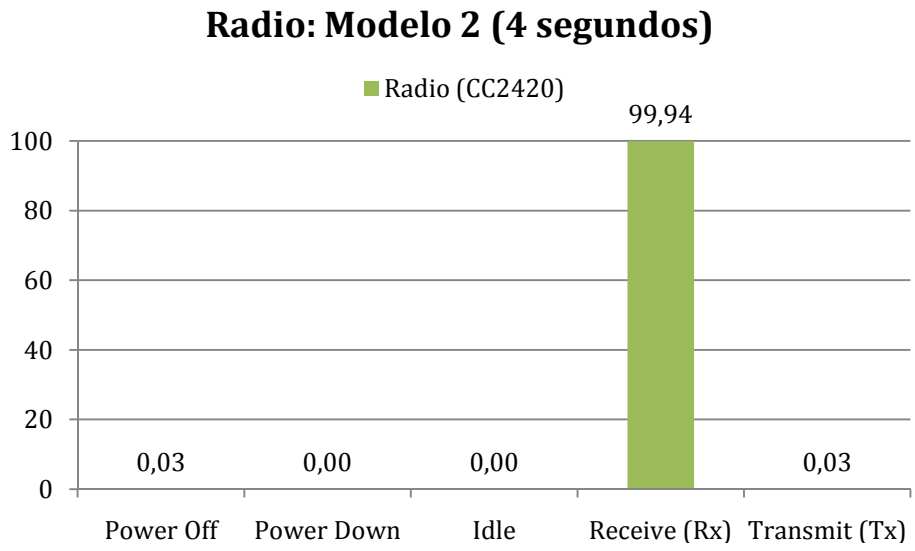


Figura 61: Porcentajes de activación de la radio (Escenario 1: Modelo 2)

En la Figura 62 se puede observar el consumo de cada componente. Como ocurre en el modelo anterior, prácticamente la totalidad del consumo es provocado por dos componentes: el microprocesador y la radio.

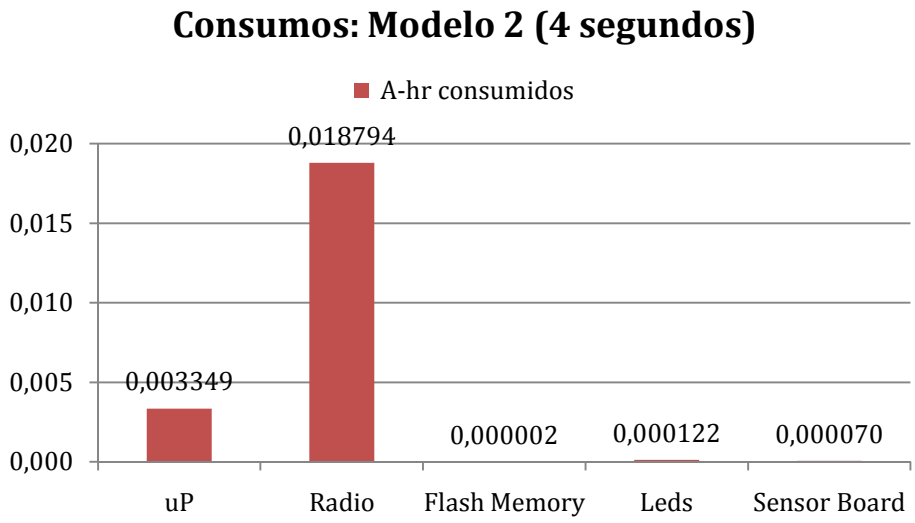


Figura 62: Consumos de los componentes (Escenario 1: Modelo 2)

8.1.1.3 Modelo 3: 8 segundos

Tras la simulación, el resultado del análisis energético que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=======================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.150559783511234 Joule
   Active: 0.07689565892277019 Joule, 24975058 cycles
   Idle: 36.07366412458846 Joule, 26517104942 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97441404188177 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.94474040917967 Joule, 26529627342 cycles
   Transmit (Tx): 0.029673211669921876 Joule, 4419690 cycles
```

En la Figura 63 muestra que en el microprocesador sigue la misma tendencia: se reduce el tiempo en el que el microprocesador está en estado `Active` y aumenta para el estado `Idle`.

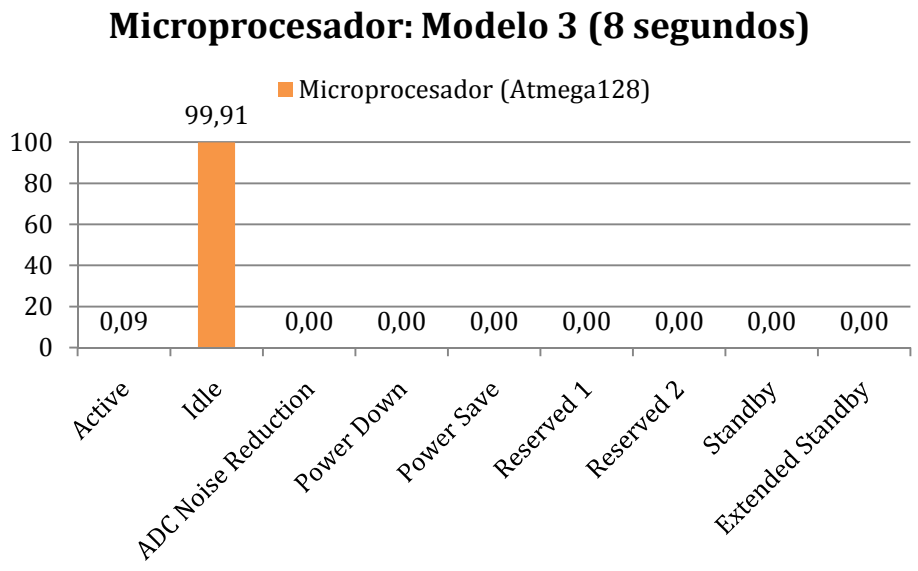


Figura 63: Porcentajes de activación de la CPU (Escenario 1: Modelo 3)

Como ocurre con el microprocesador la radio sigue la misma tendencia: se reduce el tiempo dedicado al envío de mensajes y aumenta el tiempo dedicado a la recepción de los mismos. Como puede observase en la Figura 64:

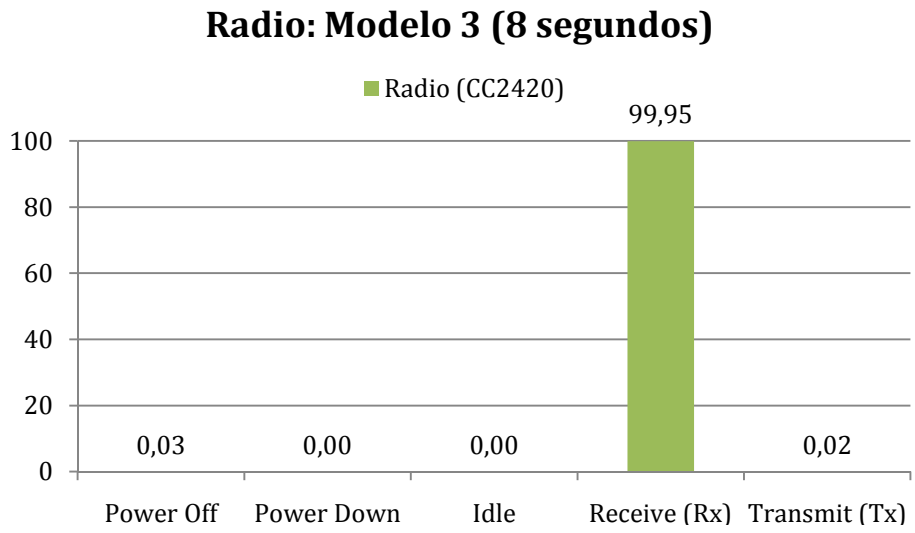


Figura 64: Porcentajes de activación de la radio (Escenario 1: Modelo 3)

En la Figura 65 muestra el consumo de cada componente. Como en los modelos anteriores, la mayor parte del consumo es provocado el microprocesador y la radio.

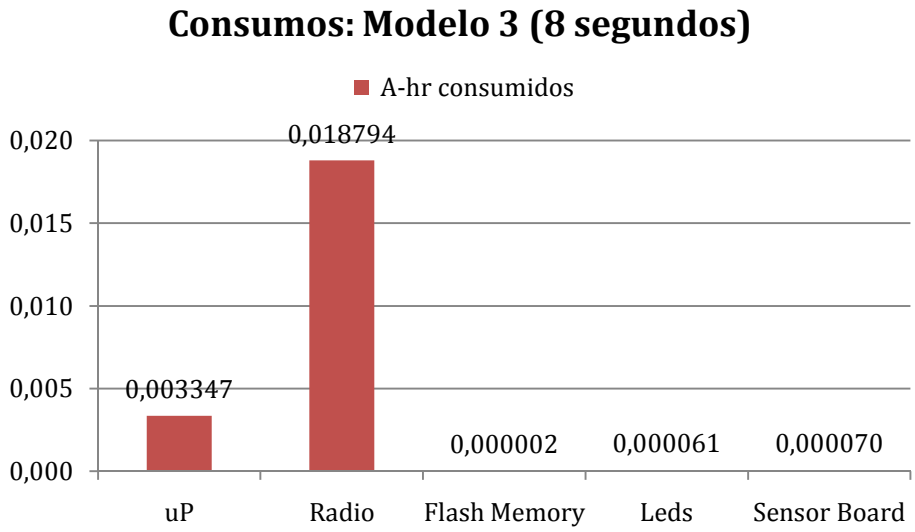


Figura 65: Consumos de los componentes (Escenario 1: Modelo 3)

8.1.1.4 Modelo 4: 16 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
==={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles,3600.0 seconds

CPU: 36.14161048380542 Joule
   Active: 0.060861973720336915 Joule, 19767453 cycles
   Idle: 36.08074851008509 Joule, 26522312547 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.9764165186884 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.96110848046874 Joule, 26531767032 cycles
   Transmit (Tx): 0.0153076171875 Joule, 2280000 cycles
```

Como muestra la Figura 66, al aumentar el intervalo de monitorización se reduce el tiempo que se encuentra el microprocesador en modo `Active` y como en los anteriores modelos, aumenta el tiempo en modo `Idle`.

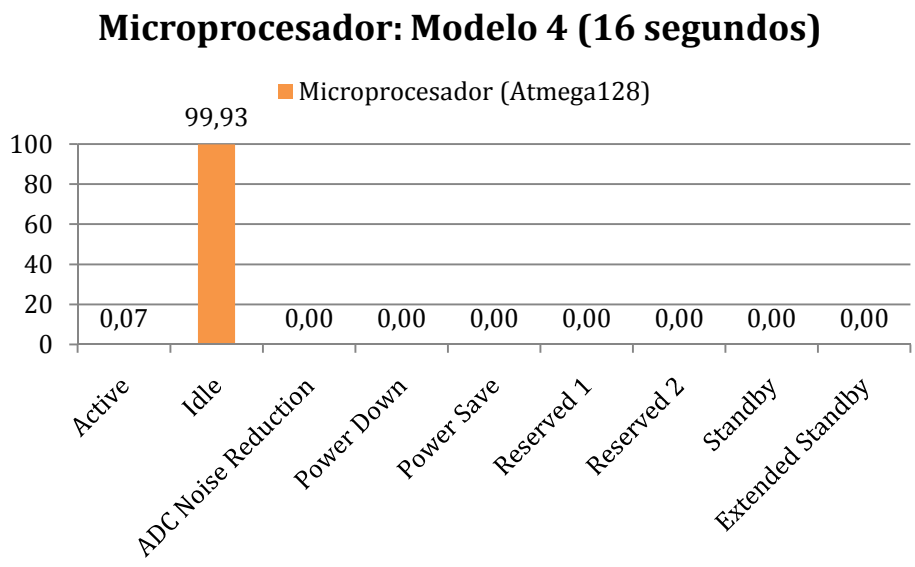


Figura 66: Porcentajes de activación de la CPU (Escenario 1: Modelo 4)

Como muestra la Figura 67, el tiempo de activación de envío de mensajes se ha reducido a la mitad, mientras que aumenta el tiempo que la radio está en modo `Receive`:

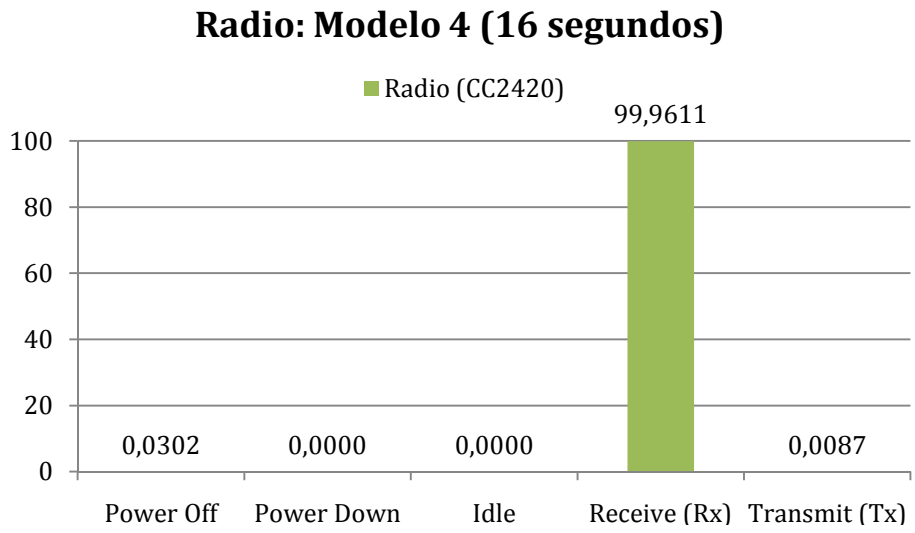


Figura 67: Porcentajes de activación de la radio (Escenario 1: Modelo 4)

En la Figura 68 se observa el consumo de cada componente del nodo. Sigue la misma tendencia que los modelos anteriores.

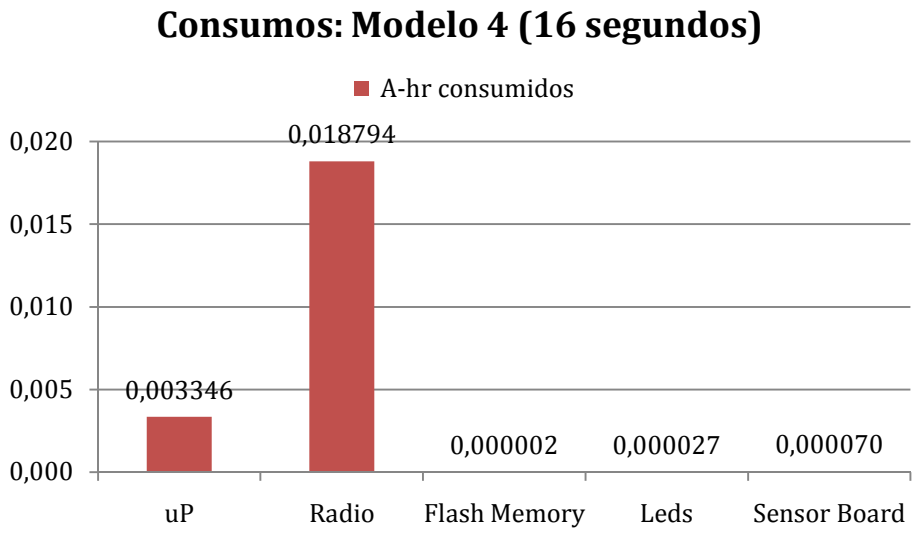


Figura 68: Consumos de los componentes (Escenario 1: Modelo 4)

8.1.1.5 Modelo 5: 32 segundos

Tras la simulación, *Avrora* muestra el siguiente análisis energético:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.13713425550488 Joule
   Active: 0.05284230315120443 Joule, 17162732 cycles
   Idle: 36.08429195235368 Joule, 26524917268 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97741775709176 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.96929251611328 Joule, 26532836877 cycles
   Transmit (Tx): 0.008124819946289063 Joule, 1210155 cycles
```

El microprocesador sigue la misma tendencia de los modelos anteriores, como puede observarse en la Figura 69, se reduce el tiempo en modo `Active` y aumenta el tiempo en modo `Idle`.

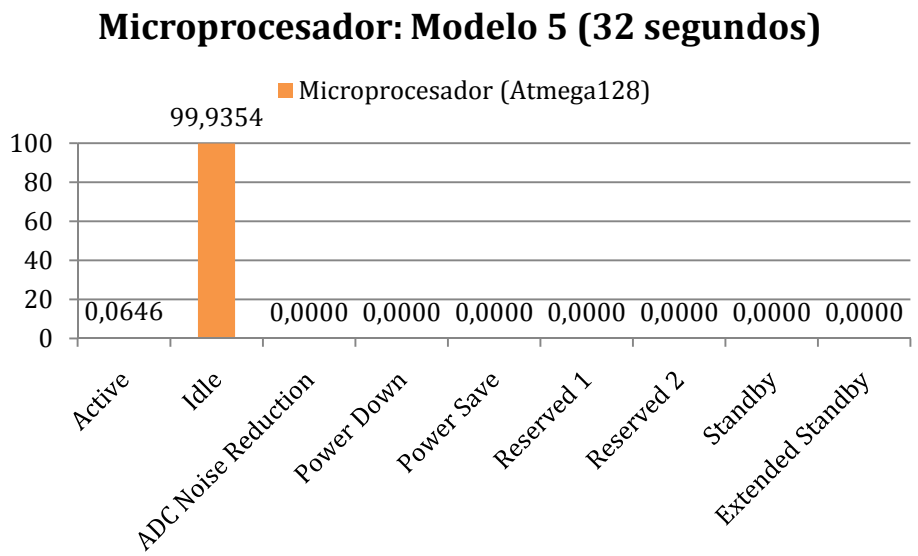


Figura 69: Porcentajes de activación de la CPU (Escenario 1: Modelo 5)

Para el caso de la radio, continua la misma tendencia que en los anteriores modelos. Es decir, el intervalo de monitorización no tiene una influencia significante en el comportamiento de la radio, como muestra la Figura 70:

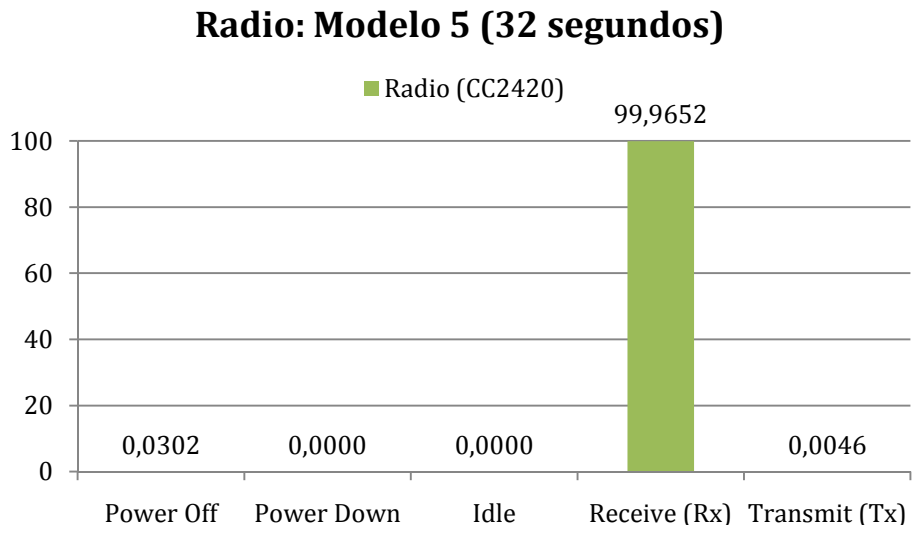


Figura 70: Porcentajes de activación de la radio (Escenario 1: Modelo 5)

En la Figura 71 muestra el consumo de los componentes ejecutando *SensitiveSampler* (CTP) con un intervalo de monitorización de 32 segundos. Parece confirmarse que dicho intervalo no influye en el comportamiento energético del nodo.

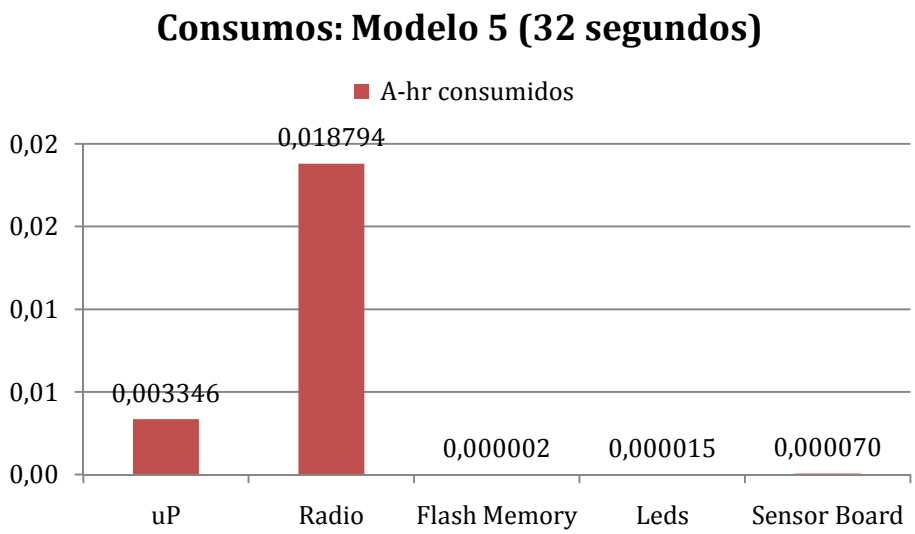


Figura 71: Consumos de los componentes (Escenario 1: Modelo 5)

8.1.1.6 Modelo 6: 64 segundos

Tras la simulación, el análisis energético que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=======================
Node lifetime: 26542080000 cycles,  3600.0 seconds

CPU: 36.134876253945556 Joule
   Active: 0.04879683731823731 Joule, 15848799 cycles
   Idle: 36.08607941662732 Joule, 26526231201 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.9779227295039 Joule
   Power Off: 6.525093587239584E-8 Joule, 8018035 cycles
   Power Down: 1.0998535156250001E-7 Joule, 13515 cycles
   Idle: 2.4579589843749997E-7 Joule, 1418 cycles
   Receive (Rx): 202.97342011669923 Joule, 26533376451 cycles
   Transmit (Tx): 0.004502191772460937 Joule, 670581 cycles
```

Como muestra la Figura 72, se puede afirmar, que para el microprocesador el intervalo de monitorización no influye en su consumo. El estado `Receive` es el que lleva el peso del gasto. Mientras el microprocesador no se encuentre en estados de mínimo consumo, esta tendencia no cambiará.

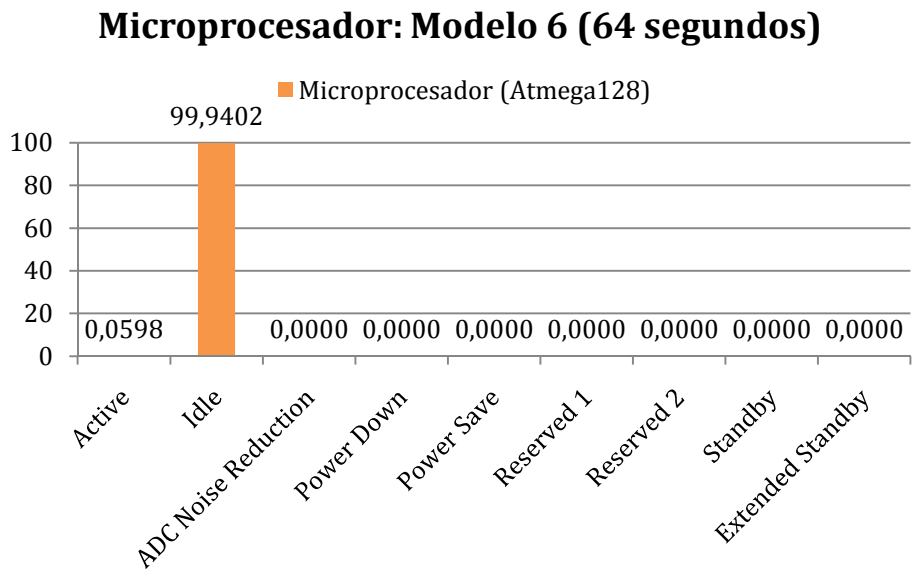


Figura 72: Porcentajes de activación de la CPU (Escenario 1: Modelo 6)

Lo mismo ocurre para la radio, mientras no se encuentre en estados de mínimo consumo (`Power Off` o `Power Down`) el intervalo de monitorización no

tendrá influencia en el comportamiento energético de la aplicación. La Figura 73 muestra claramente dicha conclusión.

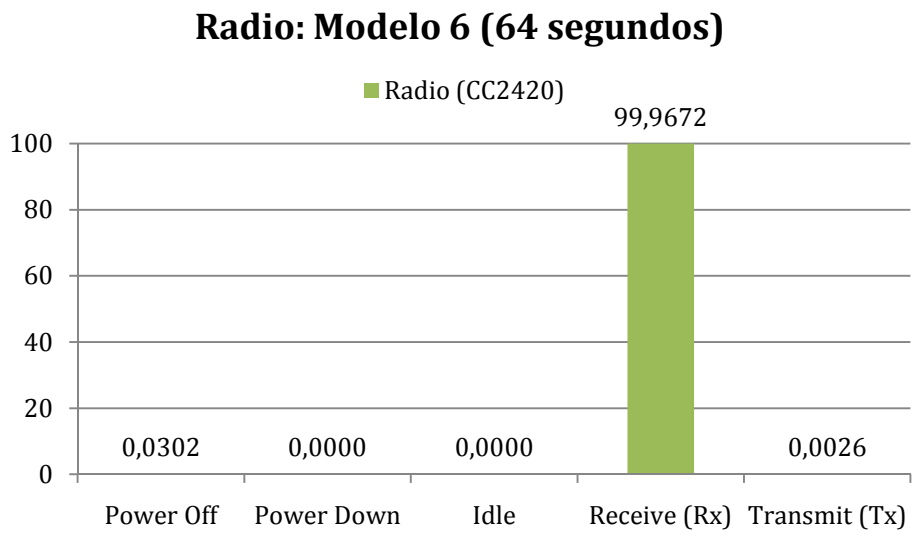


Figura 73: Porcentajes de activación de la radio (Escenario 1: Modelo 6)

Siguiendo este razonamiento el consumo del nodo (Figura 74) no cambiará significativamente con la variación del intervalo de monitorización.

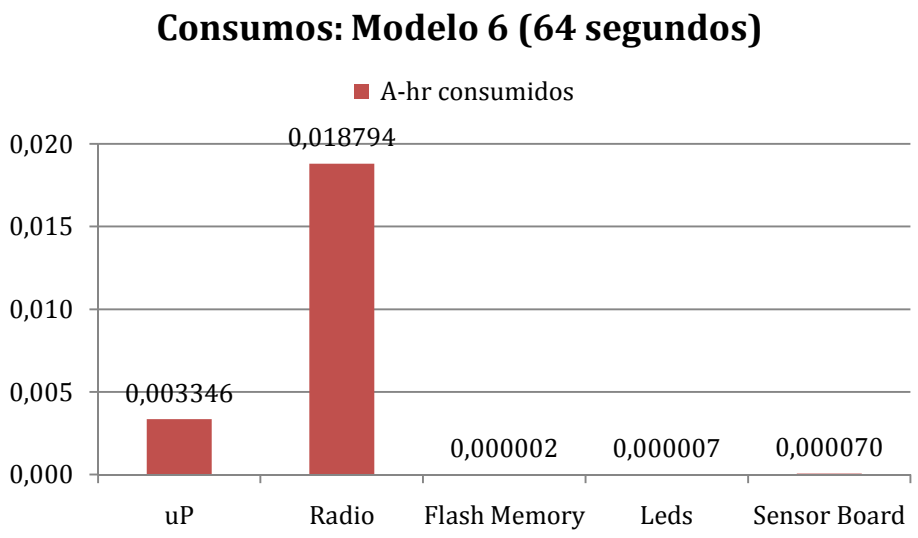


Figura 74: Consumos de los componentes (Escenario 1: Modelo 6)

8.1.1.7 Duración de baterías

En la Tabla 7 se muestra el tiempo de vida de la batería, dependiendo de su capacidad y del intervalo de monitorización empleado en cada modelo. El cálculo del tiempo de vida esperado ha sido realizado usando el modelo explicado en el

Capítulo 7. La tabla muestra que los resultados son muy similares entre los distintos modelos, la razón sigue siendo la misma, el consumo lo realiza la radio en estado de escucha mayoritariamente.

Tabla 7: Duración de las baterías (Escenario 1)

Capacidad batería (mA)	Modelo 1 (meses)	Modelo 2 (meses)	Modelo 3 (meses)	Modelo 4 (meses)	Modelo 5 (meses)	Modelo 6 (meses)
500	0,0305	0,0307	0,0308	0,0308	0,0308	0,0308
1000	0,0610	0,0613	0,0615	0,0616	0,0616	0,0617
1500	0,0915	0,0920	0,0923	0,0924	0,0924	0,0925
2000	0,1220	0,1227	0,1230	0,1232	0,1233	0,1233
2500	0,1525	0,1533	0,1538	0,1540	0,1541	0,1541
3000	0,1830	0,1840	0,1845	0,1848	0,1849	0,1850

En la Figura 75 se muestra una gráfica que compara el tiempo de vida del nodo con la capacidad de la batería para cada uno de los modelos (tiempos de monitorización distintos). Puede observarse, como los valores de los distintos modelos son prácticamente iguales y por eso se agrupan en una única función lineal. El valor máximo para una batería de 3000 mAh es de 0.18 meses, que transformado en días son 5 días y medio. Estos resultados coinciden con el estudio realizado en el artículo: *A Review of Current Operating Systems for Wireless Sensor Networks* (40). Y con el proyecto de fin de carrera: *Monitorización de la temperatura de un edificio mediante una red de sensores inalámbrica usando motes MicaZ* (41). Como puede observarse, se trata de un valor muy reducido, por lo que se hace imprescindible el uso de políticas de ahorro de energía como *Low Power Listening.*

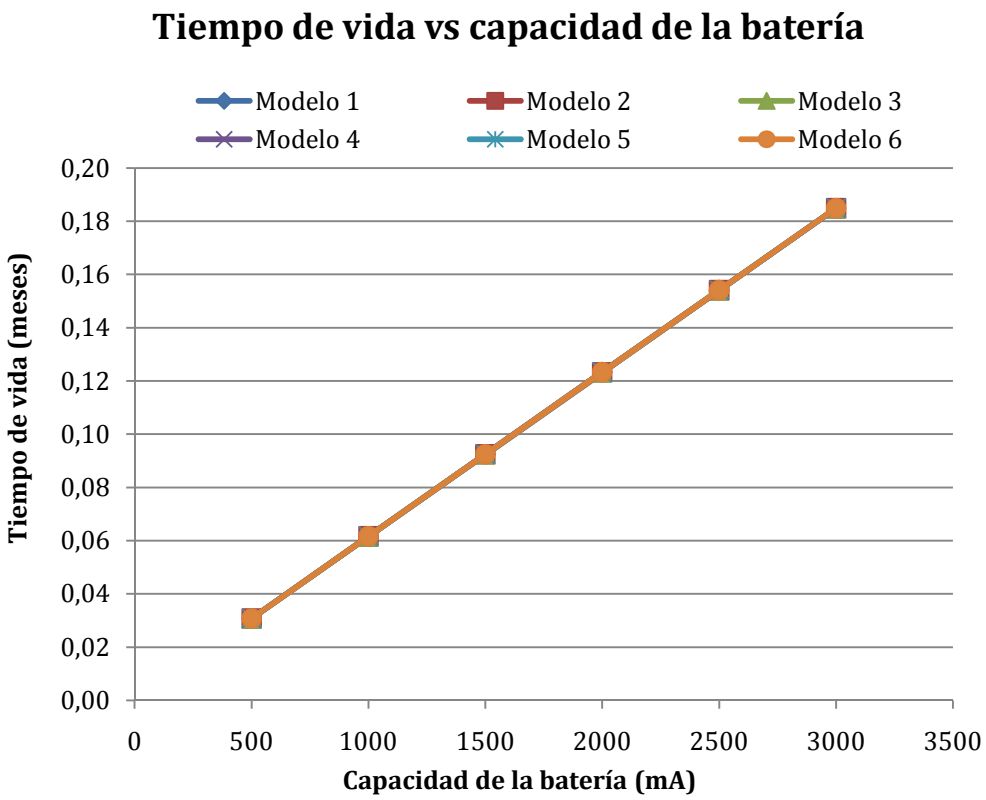


Figura 75: Gráfica tiempo de vida vs capacidad batería (Escenario 1)

La Figura 76 muestra como varia la duración de la batería de 2500 mA-h según el intervalo de monitorización empleada por la aplicación. En este caso la duración de las baterías es la misma para todos los intervalos de monitorización empleados. Esto se debe a que el consumo de la aplicación no variaba con el empleo de los deferentes intervalos.

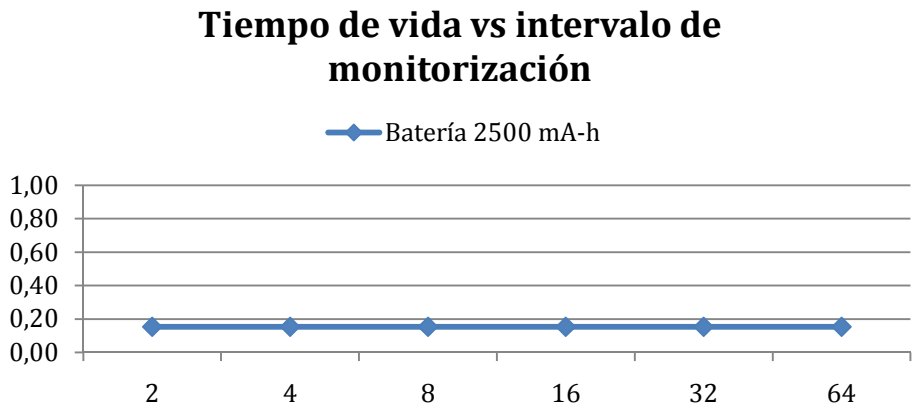


Figura 76: Gráfica tiempo de vida vs intervalo de monitorización (Escenario 1)

8.1.2 Escenario 2: Sensitive con Link Quality Indicator (LQI)

En este escenario la aplicación *SensitiveSampler* implementa el protocolo de enrutamiento LQI. Este protocolo ofrece los mismos componentes e interfaces que CTP, aunque tiene una implementación distinta. Se trata de una implementación específica para la radio CC2420. Por otro lado, LQI no soporta la implementación de *LowPowerListening*. A continuación se muestran los resultados de los consumos energéticos para cada caso.

8.1.2.1 Modelo 1: 2 segundos

El resultado que muestra *Avrora* sobre el análisis de energía es:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.1888295548169 Joule
   Active: 0.145460293704834 Joule, 47244270 cycles
   Idle: 36.04336926111206 Joule, 26494835730 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.96127487277275 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.8370018517253 Joule, 26515543391 cycles
   Transmit (Tx): 0.12427264099121094 Joule, 18509845 cycles
```

En la Figura 77 puede observarse que, como ocurre en el escenario 1, el microprocesador nunca entra en estados de mínimo consumo energético. En comparación con el escenario anterior, el microprocesador está algo menos en estado `Active` pero en un porcentaje tan bajo que casi no tiene efectos en términos de consumo.

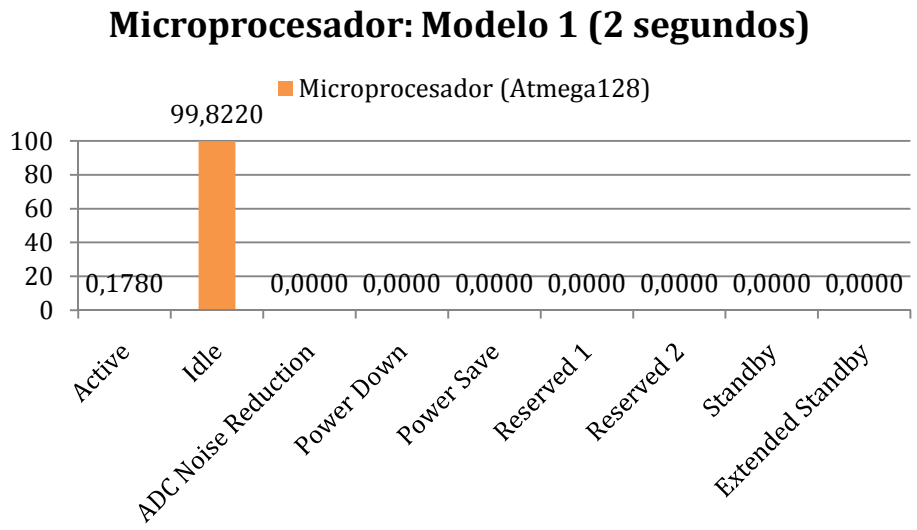


Figura 77: Porcentajes de activación de la CPU (Escenario 2: Modelo 1)

La siguiente gráfica (Figura 78) muestra, como ocurre en el escenario 1, que la radio se encuentra prácticamente la totalidad del tiempo en estado de escucha (`Receive`). El comportamiento de la radio es prácticamente el mismo que en el escenario anterior. La radio apenas está un 0,07% del tiempo enviando datos. Por otro lado, la radio prácticamente nunca se apaga, tan solo 0,03% del tiempo se encuentra en estado `Off`.

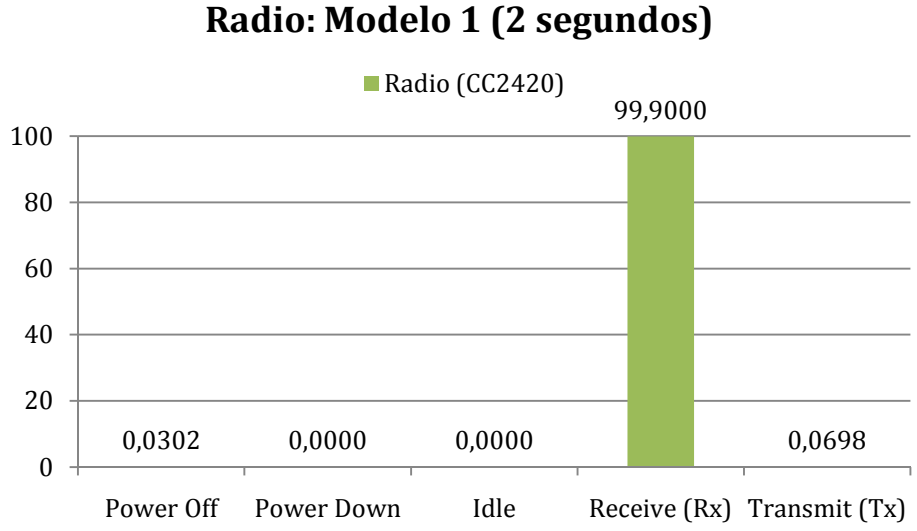


Figura 78: Porcentajes de activación de la radio (Escenario 2: Modelo 1)

En la 79 muestra el consumo en amperios de cada componente del nodo en una hora. Como ocurre en el escenario 1 (*SensitiveSampler* con CTP). La radio es el componente con mayor consumo. Esto se debe a que, como en el anterior

escenario, no se implementa *Low Power Listening* para minimizar el tiempo en modo `Receive`. El consumo del microprocesador también es elevado provocado porque la radio se encuentra escuchando en el medio.

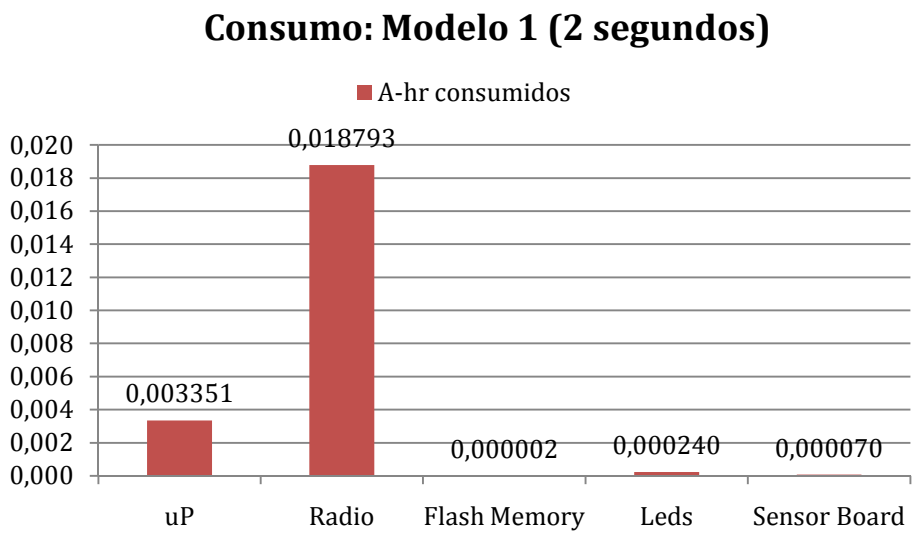


Figura 79: Consumos de los componentes (Escenario 2: Modelo 1)

8.1.2.2 Modelo 2: 4 segundos

Avrora muestra el siguiente análisis energético:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.16140167028296 Joule
   Active: 0.09632012845812989 Joule, 31283961 cycles
   Idle: 36.06508154182483 Joule, 26510796039 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.96949345614672 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.9041798375651 Joule, 26524325126 cycles
   Transmit (Tx): 0.06531323852539063 Joule, 9728110 cycles
```

Los resultados que muestra *Avrora* en el modelo 2 son muy similares a los anteriores (modelo 1). Como ocurre en el anterior escenario, el intervalo de monitorización, este no tiene una influencia significativa en los porcentajes de

tiempo en los que los componentes se encuentran en estados de consumo elevados.

La Figura 80 muestra que en el microprocesador no se producen grandes cambios en los porcentajes de activación de cada estado, a pesar de haber doblado el intervalo de monitorización con respecto al anterior modelo. Parece seguir el mismo comportamiento que en anterior escenario.

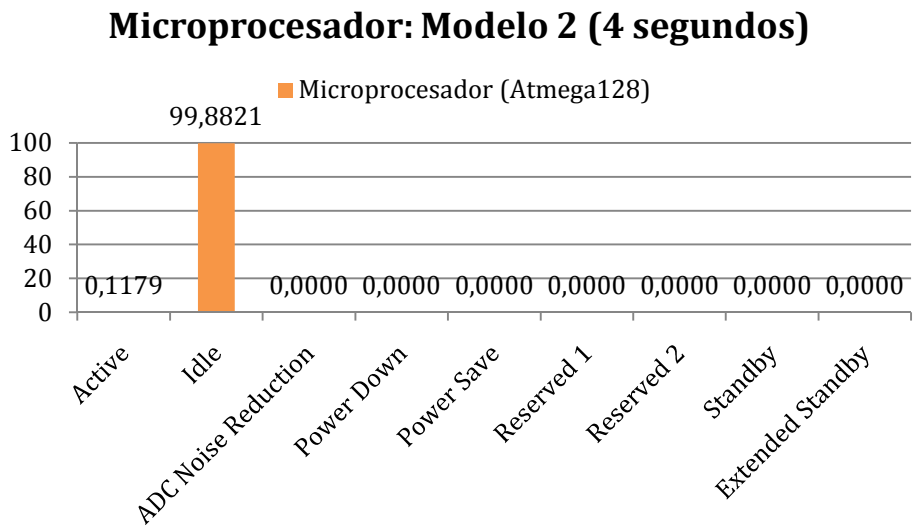


Figura 80: Porcentajes de activación de la CPU (Escenario 2: Modelo 2)

Para la radio, el envío de mensajes se ha reducido a la mitad (el tiempo de monitorización es el doble que en el anterior modelo) y se envían la mitad de paquetes de datos. Al reducirse el envió de mensajes aumenta el tiempo de recepción como puede observase en la Figura 81:

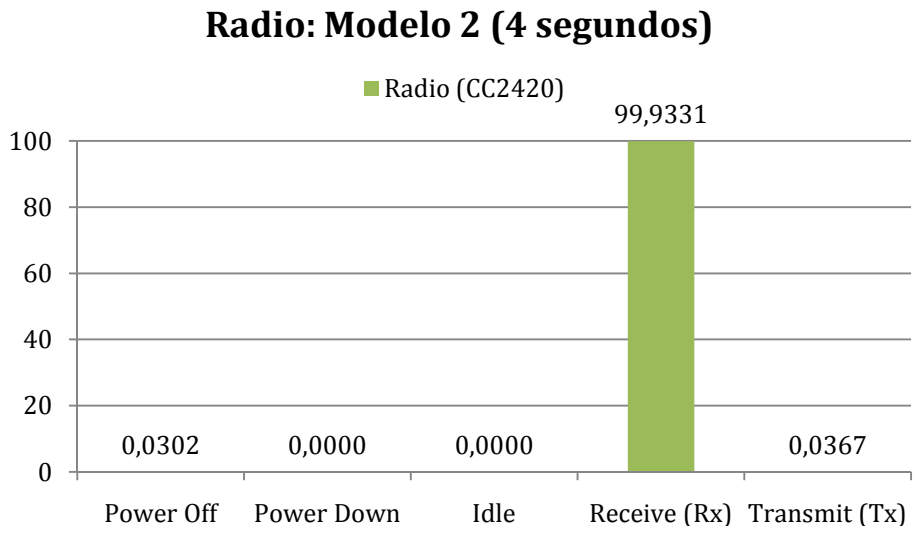


Figura 81: Porcentajes de activación de la radio (Escenario 2: Modelo 2)

En la Figura 82 se muestra el consumo de cada componente. Prácticamente la totalidad del consumo es provocado por dos componentes: el microprocesador y la radio. Parece seguir la misma tendencia que en el escenario 1.

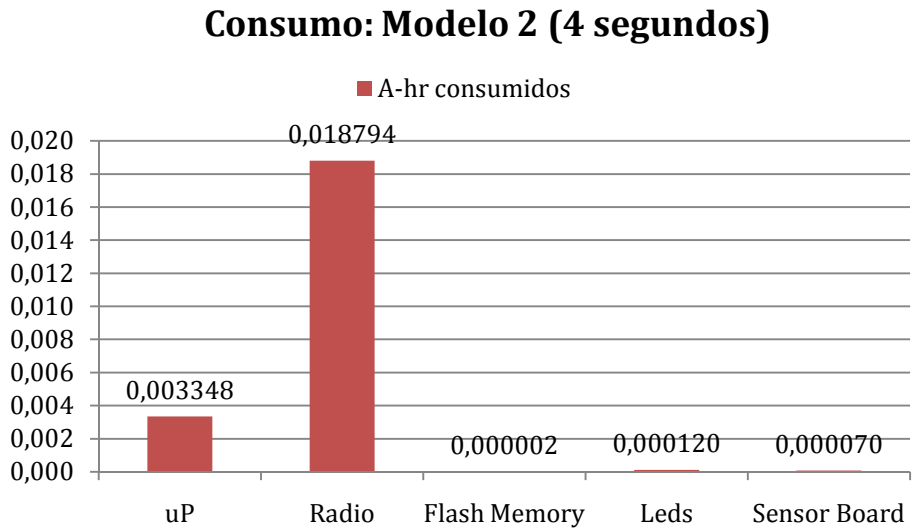


Figura 82: Consumos de los componentes (Escenario 2: Modelo 2)

8.1.2.3 Modelo 3: 8 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1
}==================================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.14744977596143 Joule
   Active: 0.07132372774715169 Joule, 23165342 cycles
   Idle: 36.07612604821428 Joule, 26518914658 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97360720960543 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.93780530061852 Joule, 26528720761 cycles
   Transmit (Tx): 0.035801528930664066 Joule, 5332475 cycles
```

La radio, como en los casos anteriores, su gasto se produce en modo escucha.

La Figura 83 muestra que el microprocesador sigue la misma tendencia: los tiempos en cada estado parecen constantes independientemente del tiempo de monitorización.

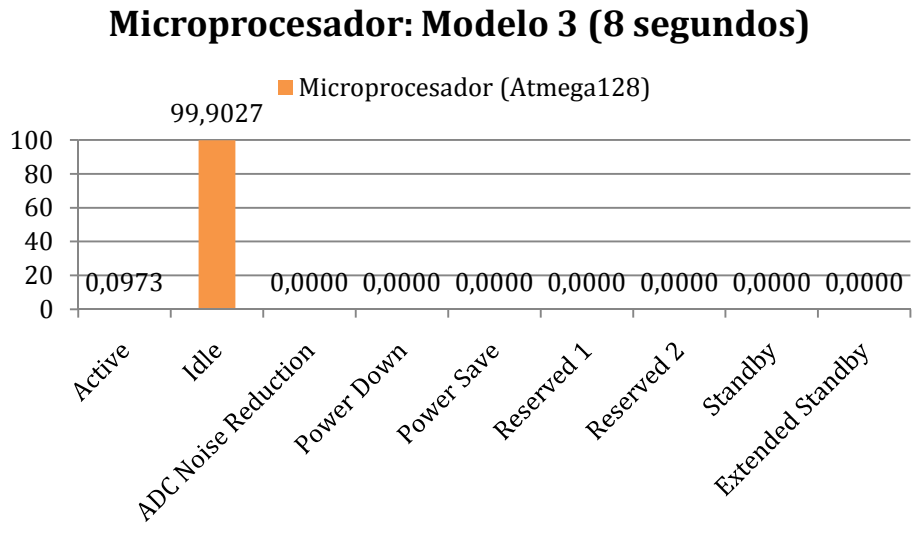


Figura 83: Porcentajes de activación de la CPU (Escenario 2: Modelo 3)

Para la radio se produce la misma tendencia: el intervalo de monitorización no influye en el comportamiento de la radio, los porcentajes de tiempo siguen el mismo patrón, como puede observase en la Figura 84:

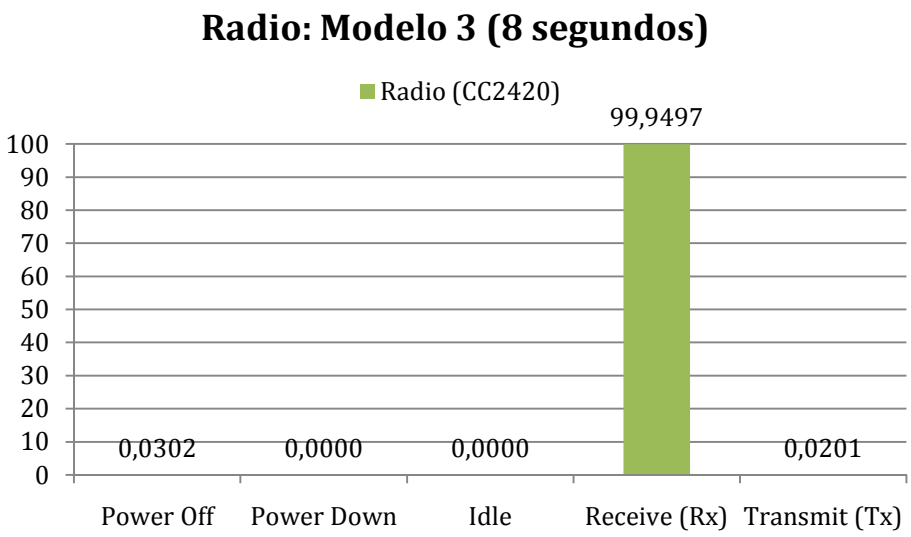


Figura 84: Porcentajes de activación de la radio (Escenario 2: Modelo 3)

La Figura 85 muestra los consumos de los componentes de un nodo. Como en los modelos anteriores y en el anterior escenario (*SensitiveSampler* con CTP), el consumo es provocado el microprocesador y la radio.

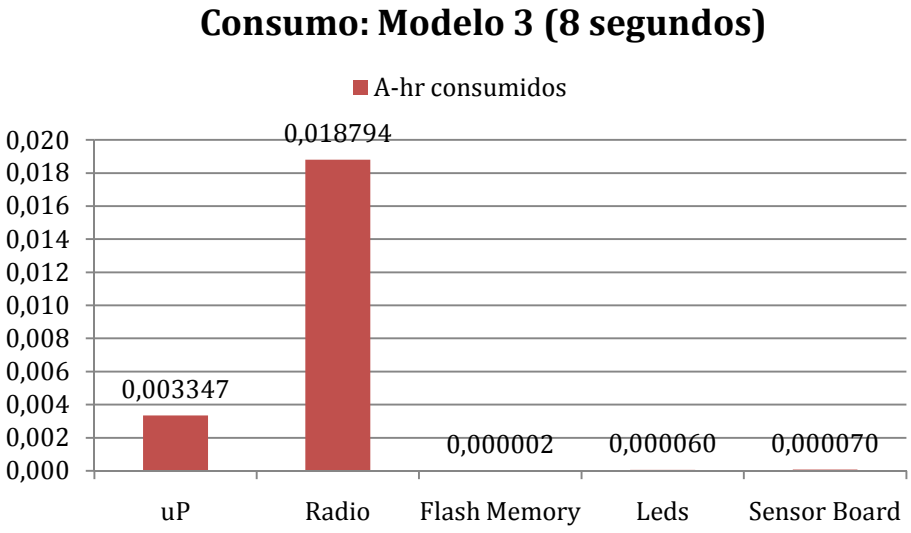


Figura 85: Consumos de los componentes (Escenario 2: Modelo 3)

8.1.2.4 Modelo 4: 16 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.14059275713941 Joule
   Active: 0.059038600996053056 Joule, 19175237 cycles
   Idle: 36.081554156143355 Joule, 26522904763 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97565962456306 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.95458156201173 Joule, 26530913811 cycles
   Transmit (Tx): 0.021077682495117187 Joule, 3139425 cycles
```

En la Figura 86 se observa que el uso de LQI como protocolo de enrutamiento no varía el comportamiento del microprocesador, ni al aumentar el intervalo de monitorización cambia el comportamiento del microprocesador.

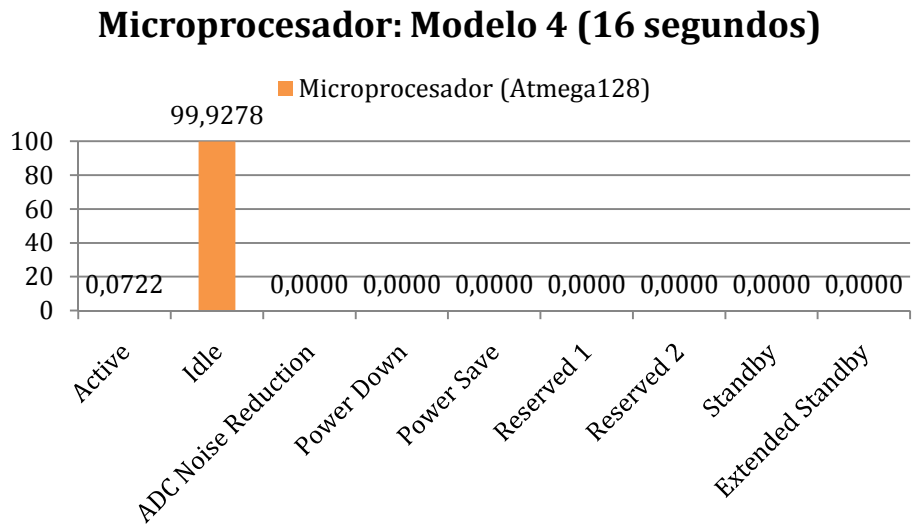


Figura 86: Porcentajes de activación de la CPU (Escenario 2: Modelo 4)

Ocurre lo mismo con la radio. La utilización de LQI no cambia el comportamiento de la radio. Como muestra la Figura 87, el tiempo de activación de envío de mensajes se ha reducido a la mitad, mientras que aumenta el tiempo que la radio esta á en modo `Receive`:

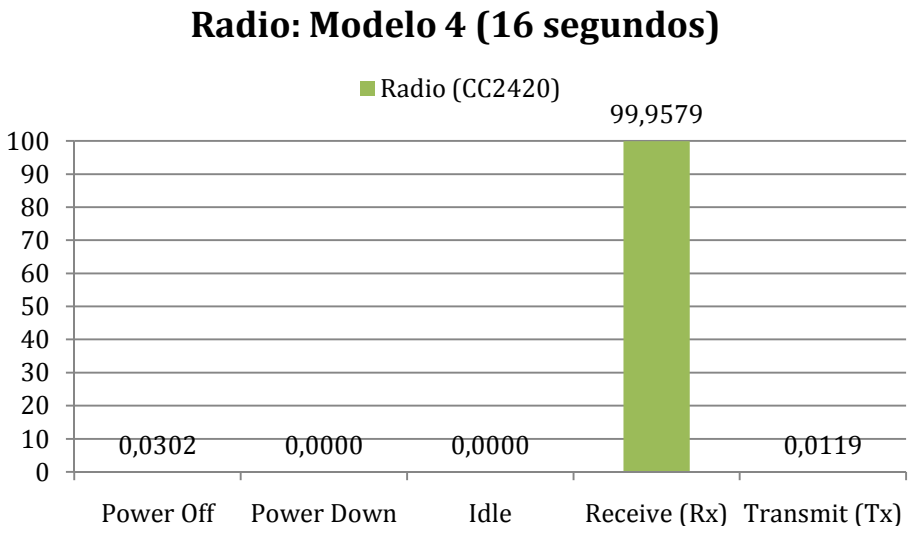


Figura 87: Porcentajes de activación de la radio (Escenario 2: Modelo 4)

En la Figura 88 se observa el consumo de cada componente del nodo. Sigue la misma tendencia que los modelos anteriores, el consumo es prácticamente el mismo en todos los casos.

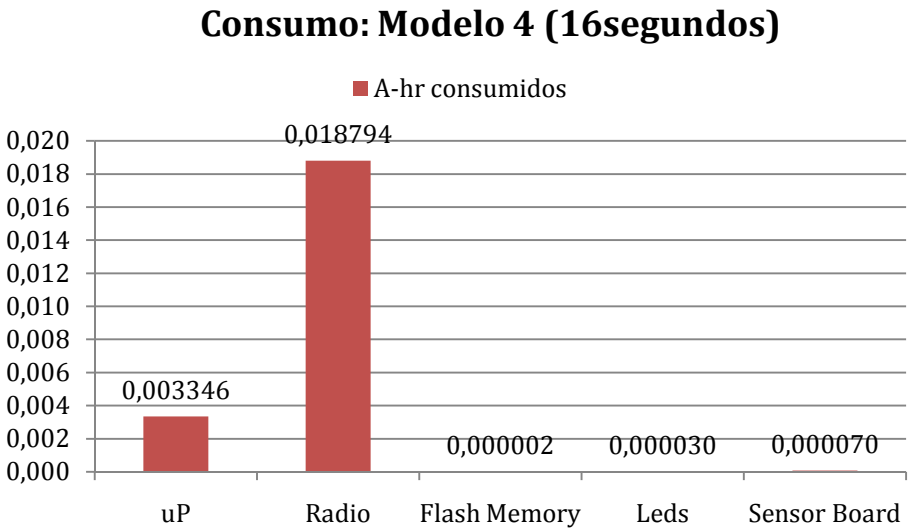


Figura 88: Consumos de los componentes (Escenario 2: Modelo 4)

8.1.2.5 Modelo 5: 32 segundos

Para este modelo Avrora muestra el siguiente análisis energético:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.13714327078662 Joule
   Active: 0.05285845505069987 Joule, 17167978 cycles
   Idle: 36.08428481573592 Joule, 26524912022 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles
   on: 5.371093750000001E-9 Joule, 6 cycles

Radio: 202.97668583204194 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.96296969270836 Joule, 26532010336 cycles
   Transmit (Tx): 0.01371575927734375 Joule, 2042900 cycles
```

La Figura 89 muestra los estados en los que se encuentra el microprocesador durante la simulación. Como en todos los casos anteriores, el comportamiento es muy parecido. El microprocesador no baja a estados en los que se reduce el consumo.

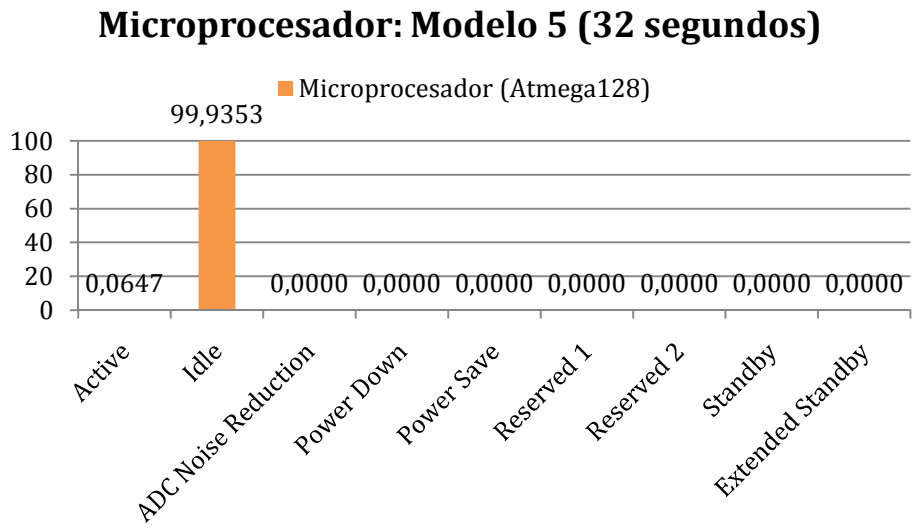


Figura 89: Porcentajes de activación de la CPU (Escenario 2: Modelo 5)

Para el caso de la radio, continua la misma tendencia que en los anteriores modelos. Es decir, el intervalo de monitorización no tiene una influencia significante en el comportamiento de la radio, como muestra la Figura 90:

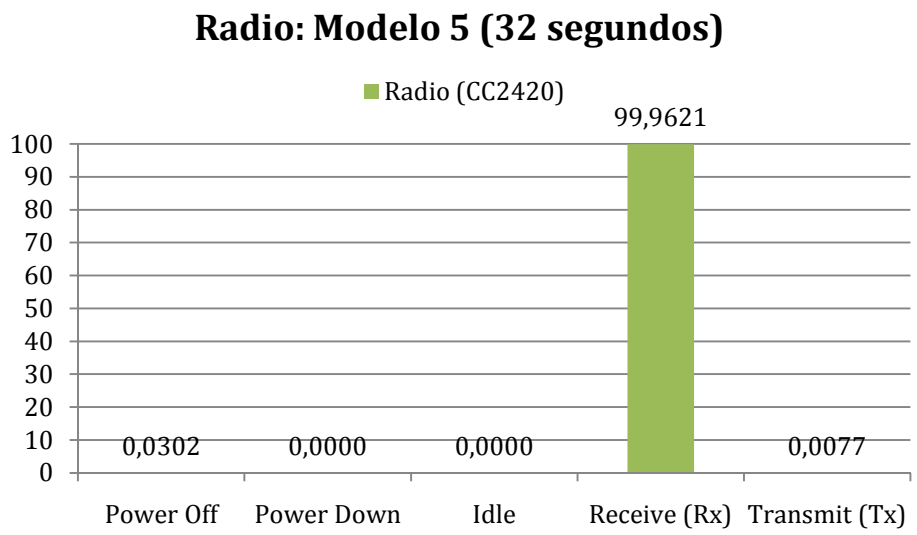


Figura 90: Porcentajes de activación de la radio (Escenario 2: Modelo 5)

En la Figura 91 se muestra el consumo de los componentes ejecutando *SensitiveSampler* con un intervalo de monitorización de 32 segundos. Parece confirmarse que la utilización de LQI no influye en comportamiento energético del nodo con respecto al escenario anterior que utilizaba CTP.

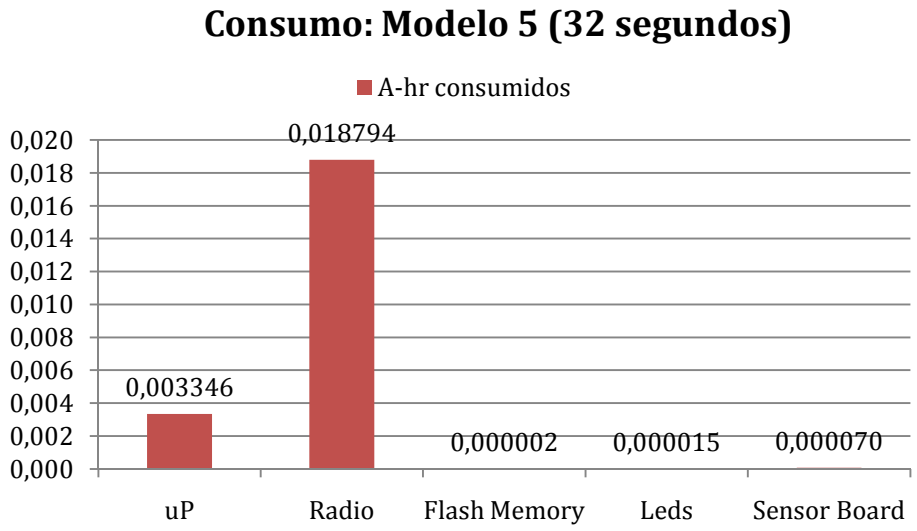


Figura 91: Consumos de los componentes (Escenario 2: Modelo 5)

8.1.2.6 Modelo 6: 64 segundos

Tras la simulación, el resultado que muestra *Avrora* para el microprocesador y la radio es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 36.13540418410108 Joule
   Active: 0.04974268405493164 Joule, 16156002 cycles
   Idle: 36.08566150004614 Joule, 26525923998 cycles
   ADC Noise Reduction: 0.0 Joule, 0 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 0.0 Joule, 0 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0 Joule, 0 cycles

Radio: 202.97720339755298 Joule
   Power Off: 6.520271809895833E-8 Joule, 8012110 cycles
   Power Down: 1.09619140625E-7 Joule, 13470 cycles
   Idle: 2.05234375E-7 Joule, 1184 cycles
   Receive (Rx): 202.9672002281901 Joule, 26532563366 cycles
   Transmit (Tx): 0.010002789306640626 Joule, 1489870 cycles
```

Como muestra la Figura 92, para el microprocesador el intervalo de monitorización no influye en su consumo. El estado `Idle` es el que lleva el peso del gasto. Respecto al escenario anterior, se confirma que el cambio de protocolo de enrutamiento no ha tenido efectos en el comportamiento del microprocesador.

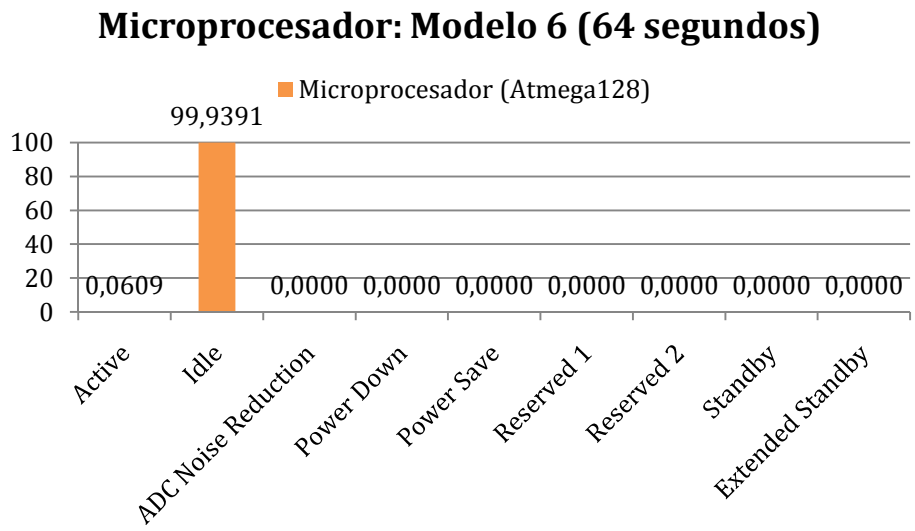


Figura 92: Porcentajes de activación de la CPU (Escenario 2: Modelo 6)

Lo mismo ocurre para la radio, mientras no se encuentre en estados de mínimo consumo (`Power Off` o `Power Down`) el intervalo de monitorización no

tendrá influencia en el comportamiento energético de la aplicación. El cambio de protocolo de enrutamiento no ha permitido que la radio tenga un comportamiento distinto. La Figura 93 muestra el comportamiento de la radio durante la simulación.

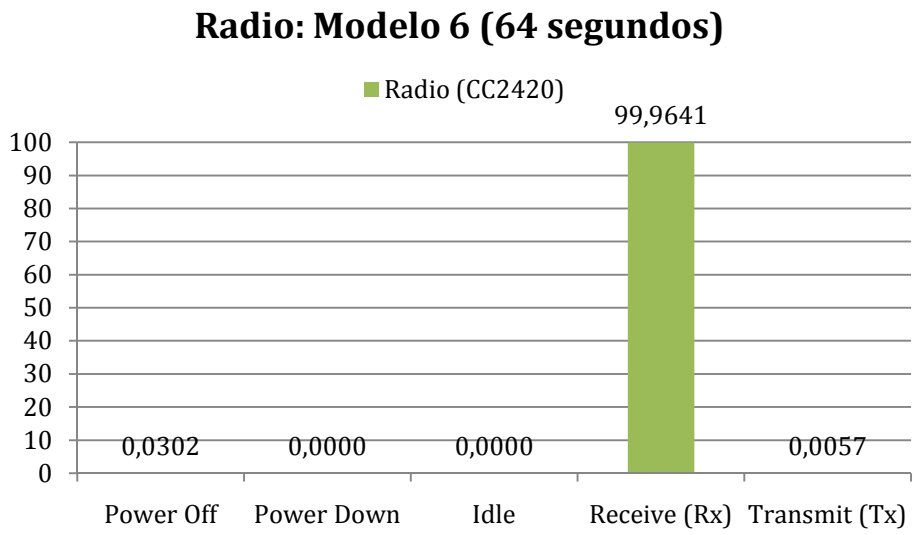


Figura 93: Porcentajes de activación de la radio (Escenario 2: Modelo 6)

Siguiendo este razonamiento el consumo del nodo (Figura 94) no cambiará significativamente con la variación del intervalo de monitorización.

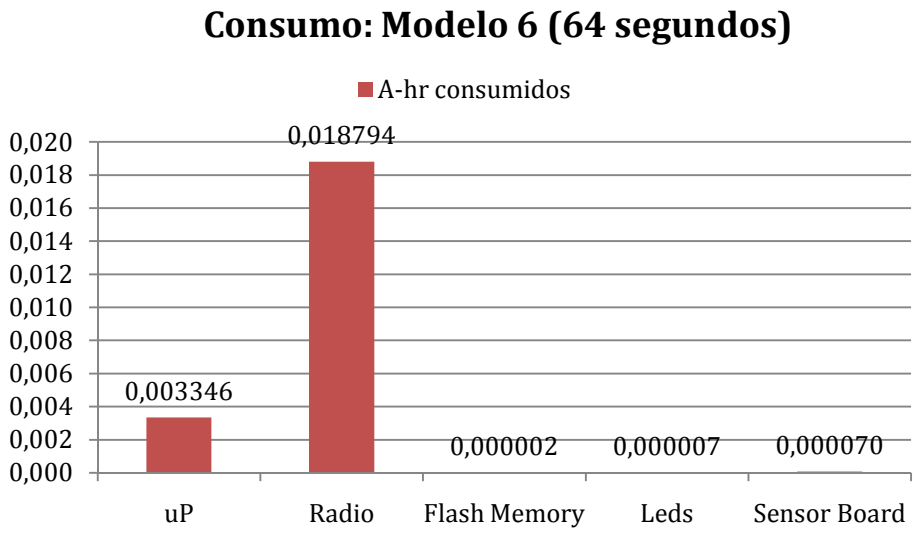


Figura 94: Consumos de los componentes (Escenario 2: Modelo 6)

8.1.2.7 Duración de baterías

En la Tabla 8 se muestra el tiempo de vida de la batería usando LQI y los distintos intervalos de monitorización, dependiendo de la capacidad de la batería.

Tabla 8: Tiempo de vida de la batería (Escenario 2)

Capacidad batería (mA)	Modelo 1 (meses)	Modelo 2 (meses)	Modelo 3 (meses)	Modelo 4 (meses)	Modelo 5 (meses)	Modelo 6 (meses)
500	0,0305	0,0307	0,0308	0,0308	0,0308	0,0308
1000	0,0610	0,0613	0,0615	0,0616	0,0616	0,0617
1500	0,0915	0,0920	0,0923	0,0924	0,0924	0,0925
2000	0,1220	0,1227	0,1230	0,1232	0,1233	0,1233
2500	0,1525	0,1533	0,1538	0,1540	0,1541	0,1541
3000	0,1830	0,1840	0,1845	0,1848	0,1849	0,1850

En la Figura 95 se muestra la gráfica que compara el tiempo de vida del nodo con la capacidad de la batería para cada uno de los intervalos de monitorización de cada modelo. Como ocurre con CTP los valores son iguales entre ellos y por eso se agrupan en una única función lineal. La aplicación implementando LQI tiene un tiempo de vida de la batería de 0.18 meses, es decir, 5.4 días. Como ocurre en el escenario 1, sería preciso el uso de políticas de ahorro de energía, pero la implementación de LQI no soporta el uso de técnicas como *LowPowerListening*. Por tanto, LQI no es un protocolo de enrutamiento adecuado para satisfacer los objetivos del proyecto.

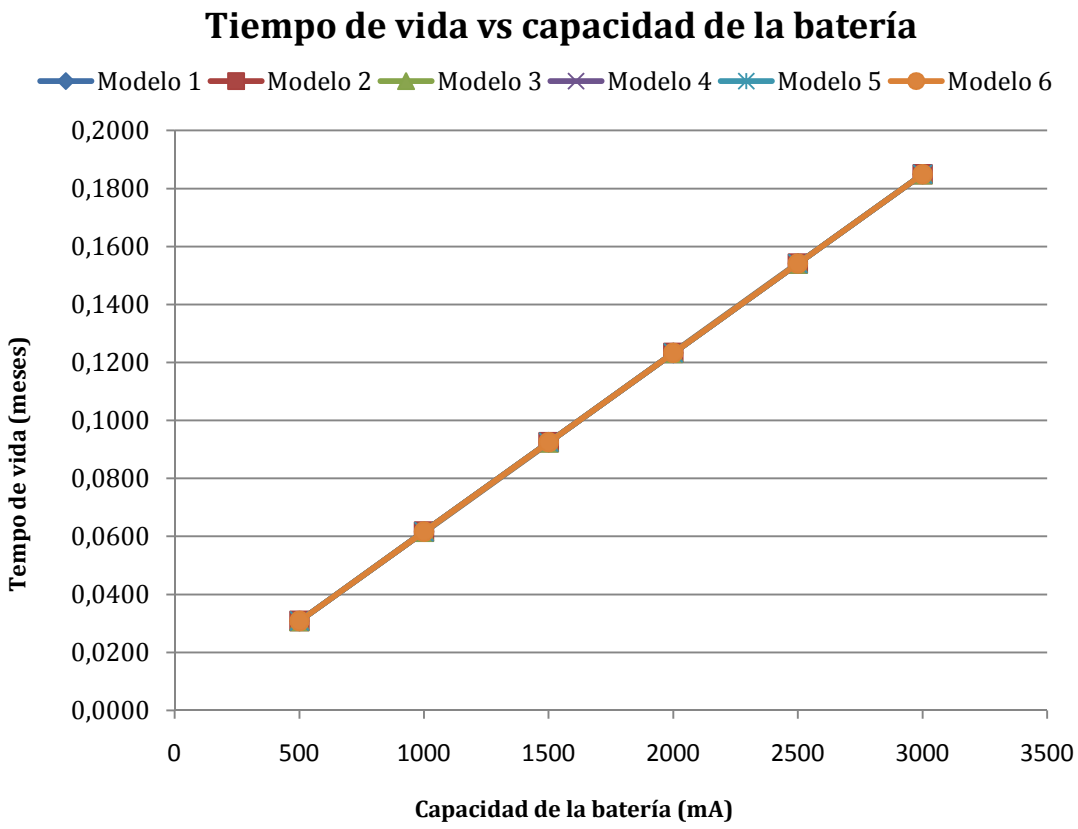


Figura 95: Gráfica duración batería vs capacidad batería (Escenario 2)

La Figura 96 muestra como varía la duración de la batería de 2500 mA-h según el intervalo de monitorización empleada por la aplicación. Al igual que ocurre en el escenario anterior, el intervalo de monitorización no afecta al tiempo de vida de la batería.

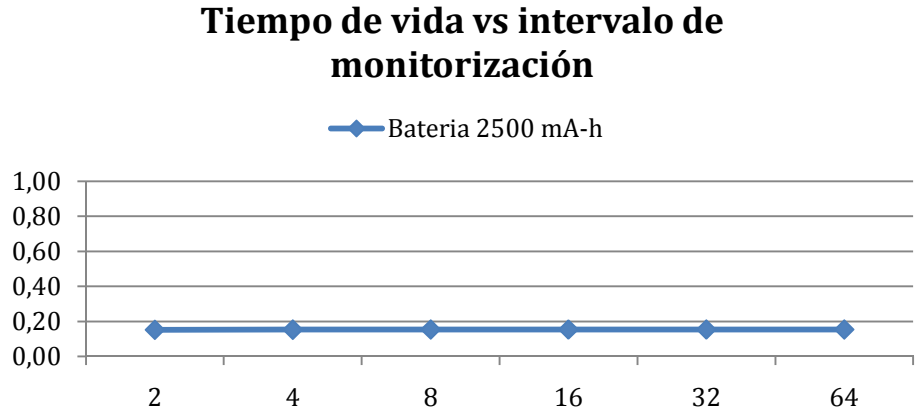


Figura 96: Gráfica tiempo de vida vs intervalo de monitorización (Escenario 2)

8.1.3 Escenario 3: *Sensitive* con Collection Tree Protocol (CTP) y Low Power Listening (LPL)

En este escenario la aplicación *SensitiveSampler* implementa el protocolo de enrutamiento *Collection Tree Protocol* y *Low Power Listening (LPL)*. Con la utilización de *LPL* se pretende minimizar el consumo de la radio apagándola el máximo tiempo posible entre una recepción de datos y la siguiente. Cuando esto ocurre la CPU puede bajar a estados de mínimo consumo, permitiendo ahorro de energía también en el microprocesador. A continuación se muestran los resultados de los consumos energéticos para cada modelo.

8.1.3.1 Modelo 1: 2 segundos

Resultado que muestra *Avrora* sobre el análisis de energía:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 6.635349830108519 Joule
   Active: 0.2852008044678141 Joule, 92630803 cycles
   Idle: 5.204689166747151 Joule, 3825873866 cycles
   ADC Noise Reduction: 5.116333393554687E-4 Joule, 1272147 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.132162902920288 Joule, 22493157237 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.012785322633911134 Joule, 129145947 cycles

Radio: 29.502872632899297 Joule
   Power Off: 1.8437281791178385E-4 Joule, 22655731865 cycles
   Power Down: 2.0333513183593753E-4 Joule, 24985821 cycles
   Idle: 4.592302880859375E-4 Joule, 2649306 cycles
   Receive (Rx): 29.386353927083338 Joule, 3841484224 cycles
   Transmit (Tx): 0.11567176757812501 Joule, 17228784 cycles
```

En la Figura 97 puede observarse que, el comportamiento del microprocesador ha cambiado considerablemente con respecto a los anteriores escenarios y se observa que el microprocesador se encuentra la mayor parte del tiempo de la simulación en estados de mínimo consumo como `Power Save`.

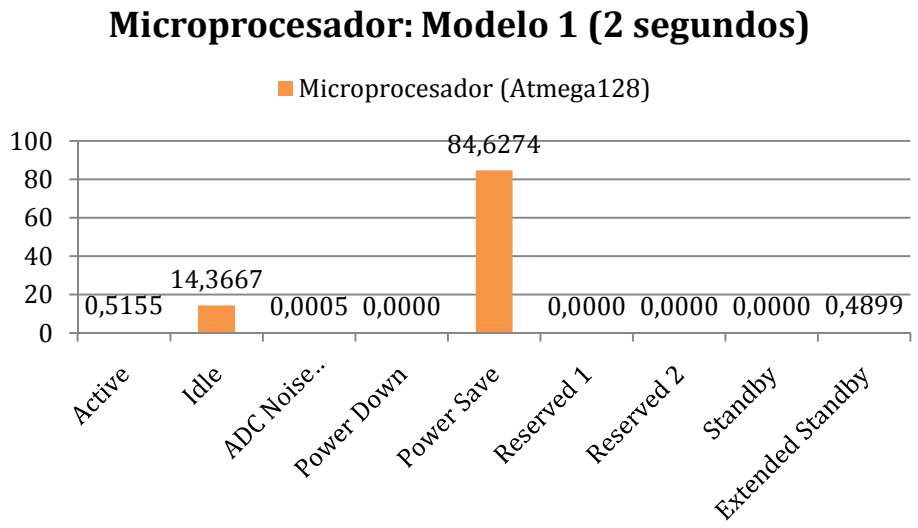


Figura 97: Porcentajes de activación de la CPU (Escenario 3: Modelo 1)

La Figura 98 muestra un comportamiento de la radio muy diferente al obtenido en anteriores escenarios. La radio se encuentra más de un 85% del tiempo de simulación en estado `Off`, y menos de un 15% en estado `Receive`. Es decir la radio se encuentra apagada el máximo posible debido a la implementación de *LowPowerListening*.

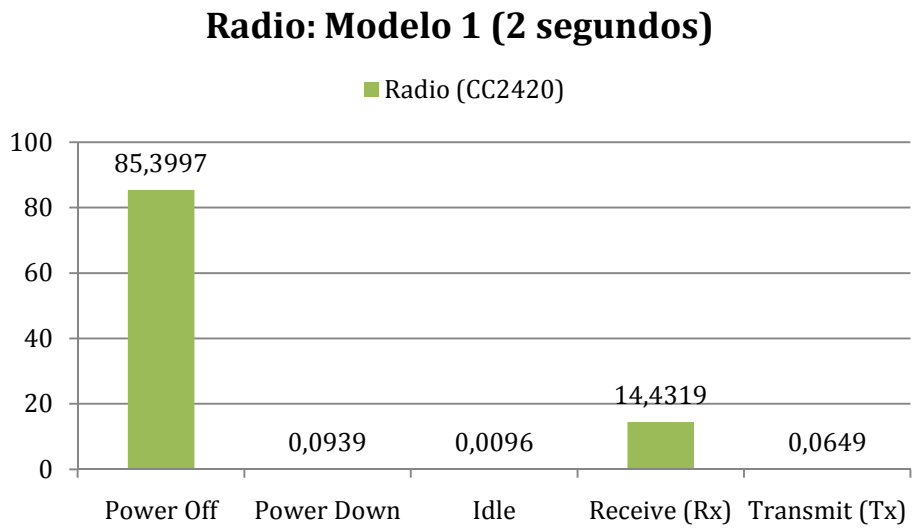


Figura 98: Porcentajes de activación de la radio (Escenario 3: Modelo 1)

En la Figura 99 se muestra el consumo en amperios de cada componente del nodo en una hora de simulación. El consumo del nodo se ha reducido drásticamente, de los 22 miliamperios de los anteriores escenarios (mismo modelo) a los actuales 3 miliamperios, es decir un 86,36%. Vemos que el ahorro se

debe a la reducción del consumo del microprocesador y de la radio ya que los demás componentes mantienen el mismo comportamiento energético.

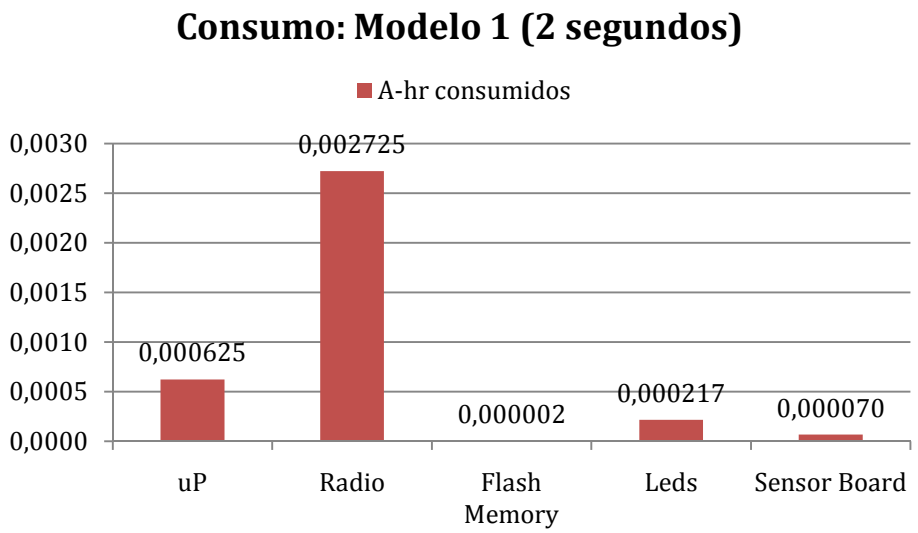


Figura 99: Consumos de los componentes (Escenario 3: Modelo 1)

8.1.3.2 Modelo 2: 4 segundos

El resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 4.021849530556925 Joule
   Active: 0.1657459371999105 Joule, 53832875 cycles
   Idle: 2.6160631189468995 Joule, 1923021183 cycles
   ADC Noise Reduction: 2.559524057617187E-4 Joule, 636411 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.2329516720730793 Joule, 24495570164 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.006832849931274414 Joule, 69019367 cycles

Radio: 14.828907130857186 Joule
   Power Off: 2.0010294865722657E-4 Joule, 24588650331 cycles
   Power Down: 1.0265116373697917E-4 Joule, 12613775 cycles
   Idle: 2.31857294921875E-4 Joule, 1337588 cycles
   Receive (Rx): 14.770038211588542 Joule, 1930789676 cycles
   Transmit (Tx): 0.05833430786132812 Joule, 8688630 cycles
```

En la Figura 100 puede observarse como ocurre otro cambio en el microprocesador en su comportamiento respecto al modelo anterior. Al aumentar el intervalo de monitorización también ha aumentado el tiempo que el microprocesador se encuentra en estado de ahorro de energía (`Power Save`).

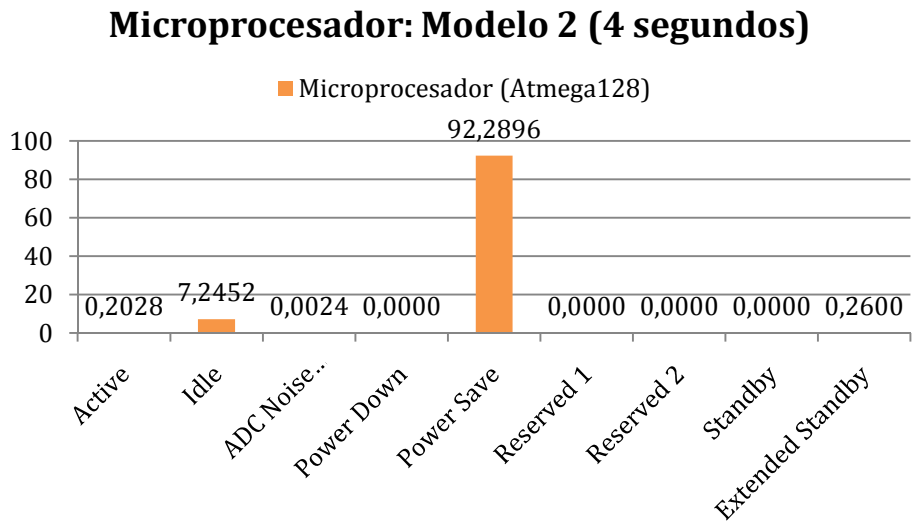


Figura 100: Porcentajes de activación de la CPU (Escenario 3: Modelo 2)

La Figura 101 muestra otro cambio en el comportamiento de la radio respecto al modelo anterior. Aumenta el tiempo que la radio se encuentra en estado `Power Off` y disminuye en el estado `Receive`. Parece que en este escenario el intervalo de monitorización sí influye en el comportamiento de la radio.

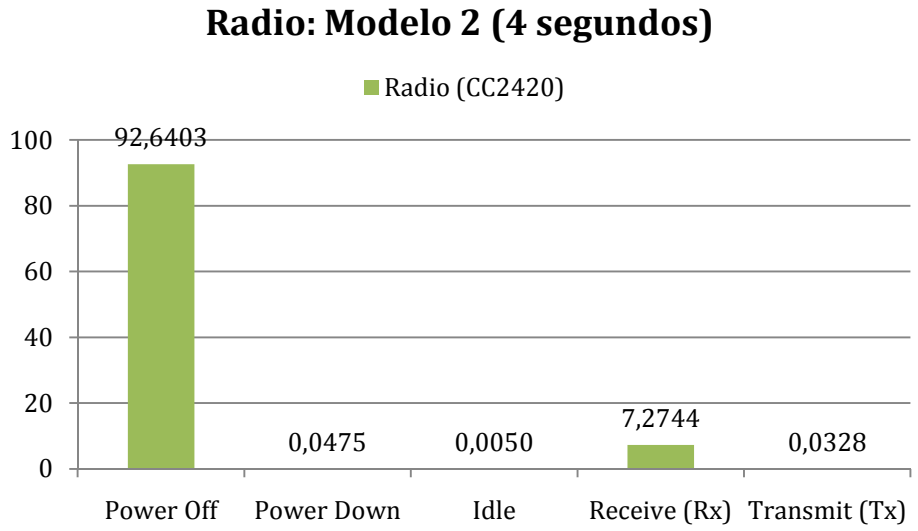


Figura 101: Porcentajes de activación de la radio (Escenario 3: Modelo 2)

En la Figura 102 se muestra el consumo de cada componente del nodo. El consumo del nodo se ha reducido respecto al modelo anterior. Parece pues, que el intervalo de monitorización influye también en el consumo de los dos principales componentes del nodo (microprocesador y radio) ya que los demás mantienen su comportamiento energético.

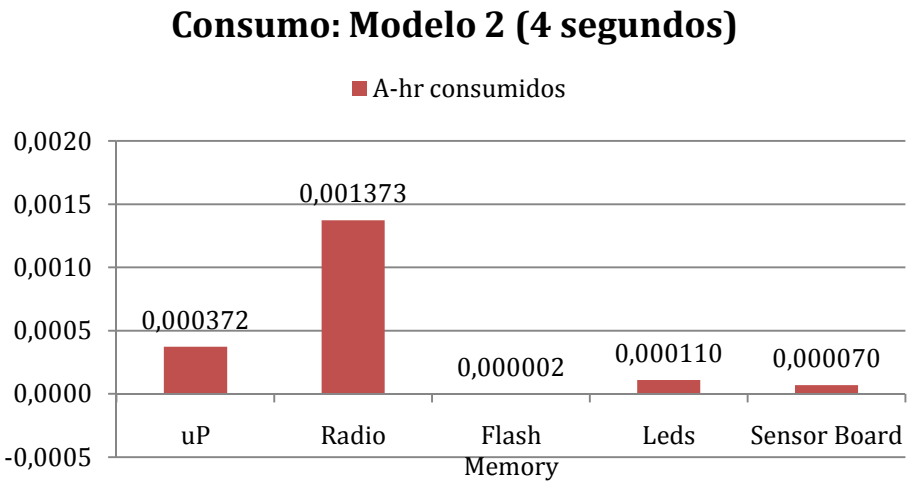


Figura 102: Consumos de los componentes (Escenario 3: Modelo 2)

8.1.3.3 Modelo 3: 8 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 2.722446823819214 Joule
   Active: 0.10624450700297038 Joule, 34507315 cycles
   Idle: 1.3292851601313884 Joule, 977133733 cycles
   ADC Noise Reduction: 1.2755934228515624E-4 Joule, 317169 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.283198772083781 Joule, 25493850463 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0035908252587890624 Joule, 36271320 cycles

Radio: 7.5347109998338135 Joule
   Power Off: 2.079226398030599E-4 Joule, 25549533979 cycles
   Power Down: 5.218695475260417E-5 Joule, 6412733 cycles
   Idle: 1.1790610839843749E-4 Joule, 680202 cycles
   Receive (Rx): 7.504729865234375 Joule, 981043836 cycles
   Transmit (Tx): 0.029603118896484374 Joule, 4409250 cycles
```

La Figura 103 muestra que el microprocesador sigue la tendencia de los anteriores modelos. Al aumentar el intervalo de monitorización también ha aumentado el tiempo que el microprocesador se encuentra en estado `Power Save`.

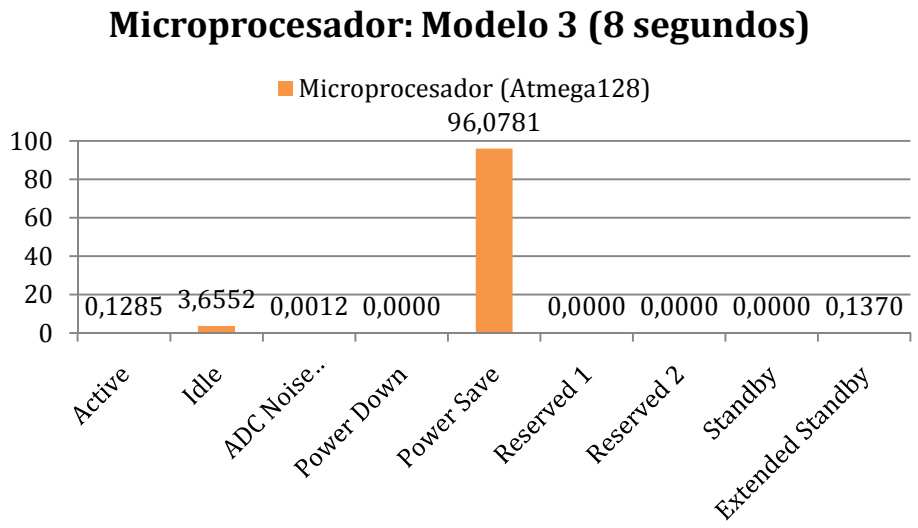


Figura 103: Porcentajes de activación de la CPU (Escenario 3: Modelo 3)

La Figura 104 muestra que la radio sigue la misma tendencia que en los modelos anteriores. Aumenta el tiempo que la radio se encuentra en estado `Power Off` y disminuye en el estado `Receive`.

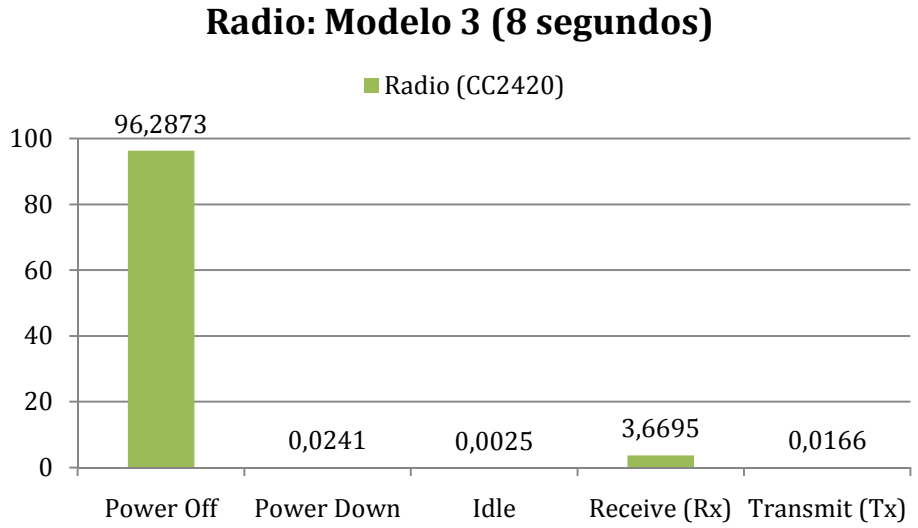


Figura 104: Porcentajes de activación de la radio (Escenario 3: Modelo 3)

En la Figura 105 se muestra el consumo total del nodo durante la simulación. El consumo se reduce prácticamente a la mitad al doblar el tiempo de monitorización.

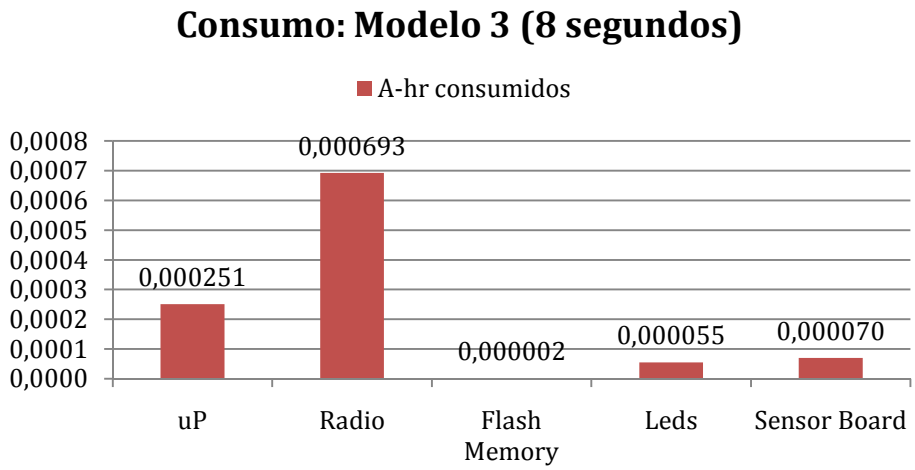


Figura 105: Consumos de los componentes (Escenario 3: Modelo 3)

8.1.3.4 Modelo 4: 16 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 2.060257407344889 Joule
   Active: 0.07878465220251464 Joule, 25588587 cycles
   Idle: 0.6705931731670328 Joule, 492941041 cycles
   ADC Noise Reduction: 6.391862467447917E-5 Joule, 158930 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.308856332890218 Joule, 26003600030 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0019593304604492187 Joule, 19791412 cycles

Radio: 3.806975820174878 Joule
   Power Off: 2.119179434326172E-4 Joule, 26040476889 cycles
   Power Down: 2.7253011067708336E-5 Joule, 3348850 cycles
   Idle: 6.154847167968751E-5 Joule, 355074 cycles
   Receive (Rx): 3.7914375763346353 Joule, 495629627 cycles
   Transmit (Tx): 0.0152375244140625 Joule, 2269560 cycles
```

La Figura 106 muestra el mismo comportamiento para el microprocesador. Al aumentar el intervalo de monitorización disminuye el tiempo que el microprocesador se encuentra en estado `Active`.

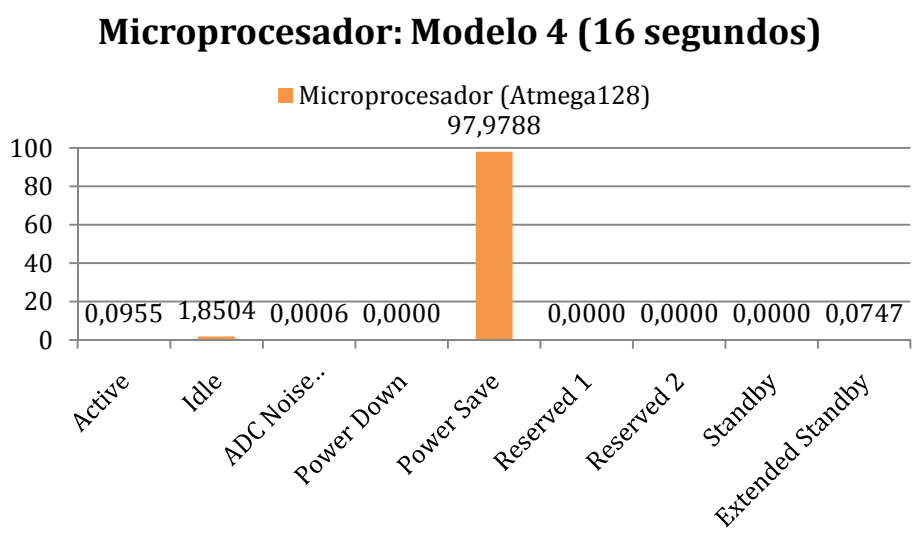


Figura 106: Porcentajes de activación de la CPU (Escenario 3: Modelo 4)

En la Figura 107 se observa que la radio sigue el mismo comportamiento que el microprocesador. Se reduce el tiempo que la radio se encuentre en modo `Receive`. Al aumentar el intervalo de monitorización también se reduce el tiempo que la radio se encuentra en estado `Transmit`.

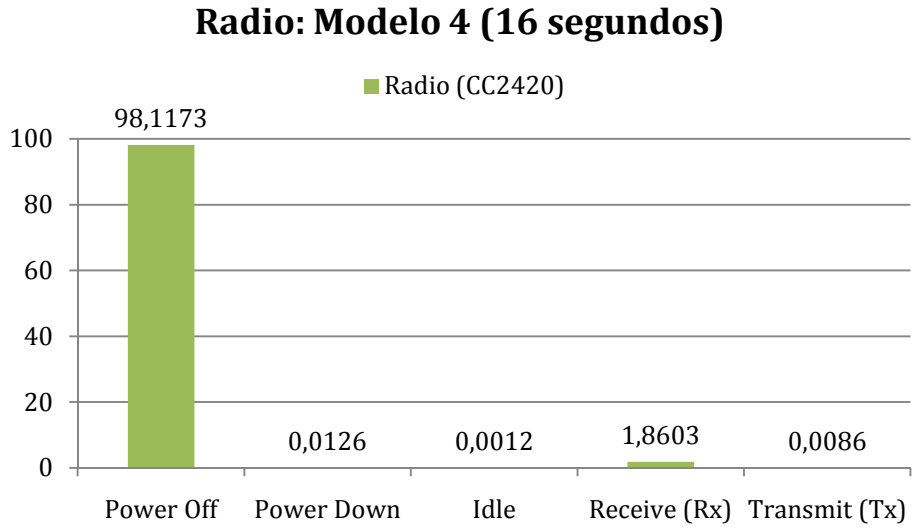


Figura 107: Porcentajes de activación de la radio (Escenario 3: Modelo 4)

En la Figura 108 muestra el consumo del nodo. Como en los modelos anteriores al doblar el intervalo de monitorización, el consumo reduce a la mitad.

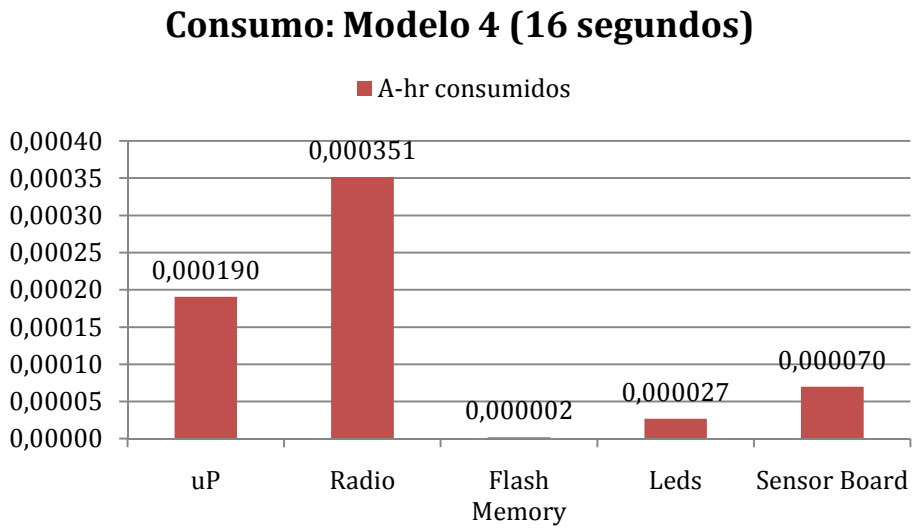


Figura 108: Consumos de los componentes (Escenario 3: Modelo 4)

8.1.3.5 Modelo 5: 32 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles,  3600.0 seconds

CPU: 1.7432491478782144 Joule
   Active: 0.0639617744715983 Joule, 20774242 cycles
   Idle: 0.35699651425443524 Joule, 262421749 cycles
   ADC Noise Reduction: 3.1959312337239585E-5 Joule, 79465 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.321125418836914 Joule, 26247355128 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 0.0011334810029296875 Joule, 11449416 cycles

Radio: 2.028960815719776 Joule
   Power Off: 2.1382427283528645E-4 Joule, 26274726646 cycles
   Power Down: 1.4775805664062501E-5 Joule, 1815651 cycles
   Idle: 3.33696533203125E-5 Joule, 192510 cycles
   Receive (Rx): 2.0206441188151043 Joule, 264145478 cycles
   Transmit (Tx): 0.008054727172851562 Joule, 1199715 cycles
```

La Figura 109 muestra los porcentajes de activación de cada estado del microprocesador. Al aumentar el intervalo de monitorización disminuye el tiempo que el microprocesador se encuentra en estado `Active` y aumenta el tiempo que se encuentra en estado `Power Save` que roza el 99% del tiempo.

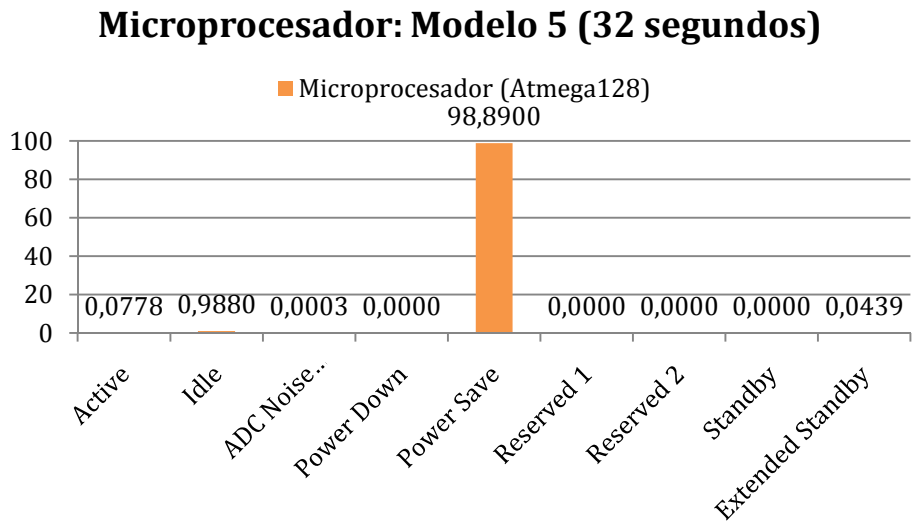


Figura 109: Porcentajes de activación de la CPU (Escenario 3: Modelo 5)

En la Figura 110 muestra los porcentajes de activación de los distintos estados de la radio. La figura muestra como se reduce el tiempo que la radio se encuentre en modo `Receive`, que se encuentra por debajo del 1%.

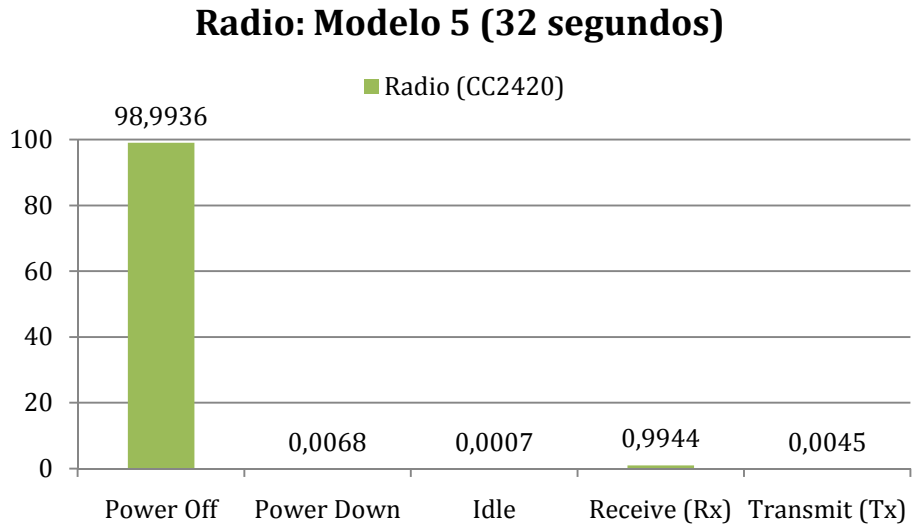


Figura 110: Porcentajes de activación de la radio (Escenario 3: Modelo 5)

En la Figura 111 muestra el consumo del nodo. El consumo de tanto de la radio como del microprocesador se ha reducido tanto que el consumo simulado para la placa sensora toma importancia respecto al consumo total del nodo.

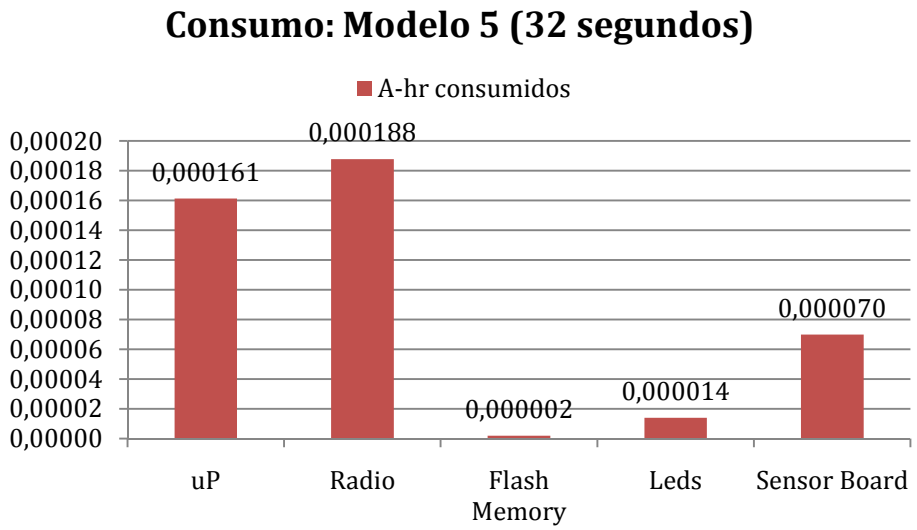


Figura 111: Consumos de los componentes (Escenario 3: Modelo 5)

8.1.3.6 Modelo 6: 64 segundos

Tras la simulación, el resultado que muestra *Avrora* es el siguiente:

```
=={ Energy consumption results for node 1 }=========================
Node lifetime: 26542080000 cycles, 3600.0 seconds

CPU: 1.5732965186130776 Joule
   Active: 0.056446187259155275 Joule, 18333243 cycles
   Idle: 0.18838947125882974 Joule, 138481729 cycles
   ADC Noise Reduction: 1.584070263671875E-5 Joule, 39387 cycles
   Power Down: 0.0 Joule, 0 cycles
   Power Save: 1.3276718437513426 Joule, 26377415709 cycles
   RESERVED 1: 0.0 Joule, 0 cycles
   RESERVED 2: 0.0 Joule, 0 cycles
   Standby: 0.0 Joule, 0 cycles
   Extended Standby: 7.731756411132812E-4 Joule, 7809932 cycles

Radio: 1.0734491841554281 Joule
   Power Off: 2.1484831883951822E-4 Joule, 26400561419 cycles
   Power Down: 8.4208984375E-6 Joule, 1034760 cycles
   Idle: 1.90330615234375E-5 Joule, 109802 cycles
   Receive (Rx): 1.0687747828776042 Joule, 139713878 cycles
   Transmit (Tx): 0.0044320989990234375 Joule, 660141 cycles
```

La Figura 112 muestra los porcentajes de activación de cada estado del microprocesador. Se confirma la tendencia que se mostraba en los anteriores modelos. El intervalo influye en el comportamiento energético del microprocesador.

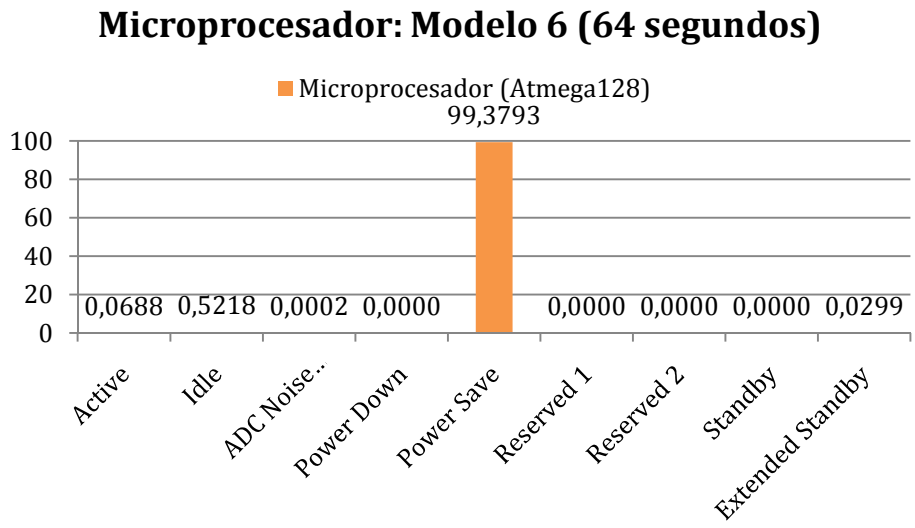


Figura 112: Porcentajes de activación de la CPU (Escenario 3: Modelo 6)

En la Figura 113 se muestran los porcentajes de activación de los distintos estados de la radio. Se confirma la tendencia que se mostraba en los anteriores modelos. El intervalo influye en el comportamiento energético de la radio.

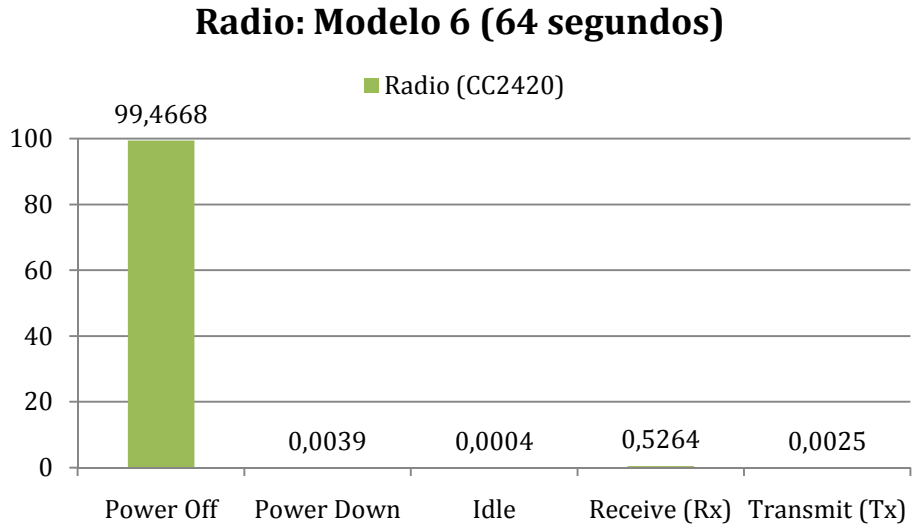


Figura 113: Porcentajes de activación de la radio (Escenario 3: Modelo 6)

En la Figura 114 se muestra el consumo del nodo. Se confirma la tendencia que se mostraba en los anteriores modelos. El intervalo de monitorización influye ahora en el comportamiento energético del nodo. Cuanto mayor sea el intervalo de monitorización menor será el consumo de la aplicación y por tanto sus baterías tendrán una mayor duración.

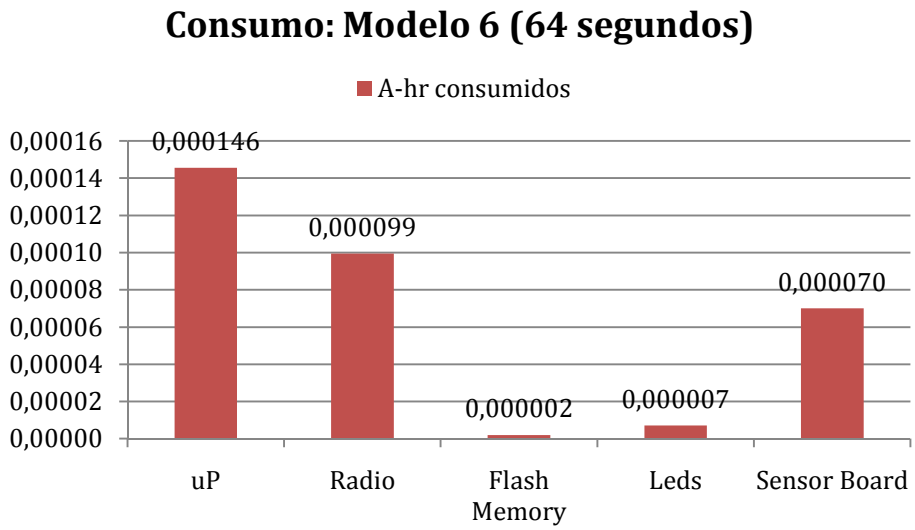


Figura 114: Consumos de los componentes (Escenario 3: Modelo 6)

8.1.3.7 Duración de baterías

En la Tabla 9 se muestra el tiempo de vida de la batería y por tanto del nodo sensor. Dependiendo de la capacidad de la batería y del intervalo de monitorización (modelo) empleado en cada caso. Los resultados son más que satisfactorios pasando, en escenarios anteriores, de 5 días y medio a más de un año utilizando *Low Power Listening*.

Tabla 9: Tiempo de vida de la batería (Escenario 3)

Capacidad batería (mA)	Modelo 1 (meses)	Modelo 2 (meses)	Modelo 3 (meses)	Modelo 4 (meses)	Modelo 5 (meses)	Modelo 6 (meses)
500	0,1882	0,3553	0,6394	1,0696	1,5739	2,1128
1000	0,3765	0,7106	1,2788	2,1391	3,1478	4,2257
1500	0,5647	1,0659	1,9182	3,2087	4,7217	6,3385
2000	0,7530	1,4213	2,5576	4,2783	6,2955	8,4514
2500	0,9412	1,7766	3,1970	5,3479	7,8694	10,5642
3000	1,1294	2,1319	3,8364	6,4174	9,4433	12,6770

En la Figura 115 se muestra la gráfica que compara el tiempo de vida del nodo con la capacidad de la batería para cada uno de los intervalos de monitorización utilizados en cada modelo. Es posible ver el diferente comportamiento entre los distintos modelos. Todos se comportan de una forma lineal respecto a la capacidad de la batería ampliándose las diferencias cuanto mayor es la capacidad.

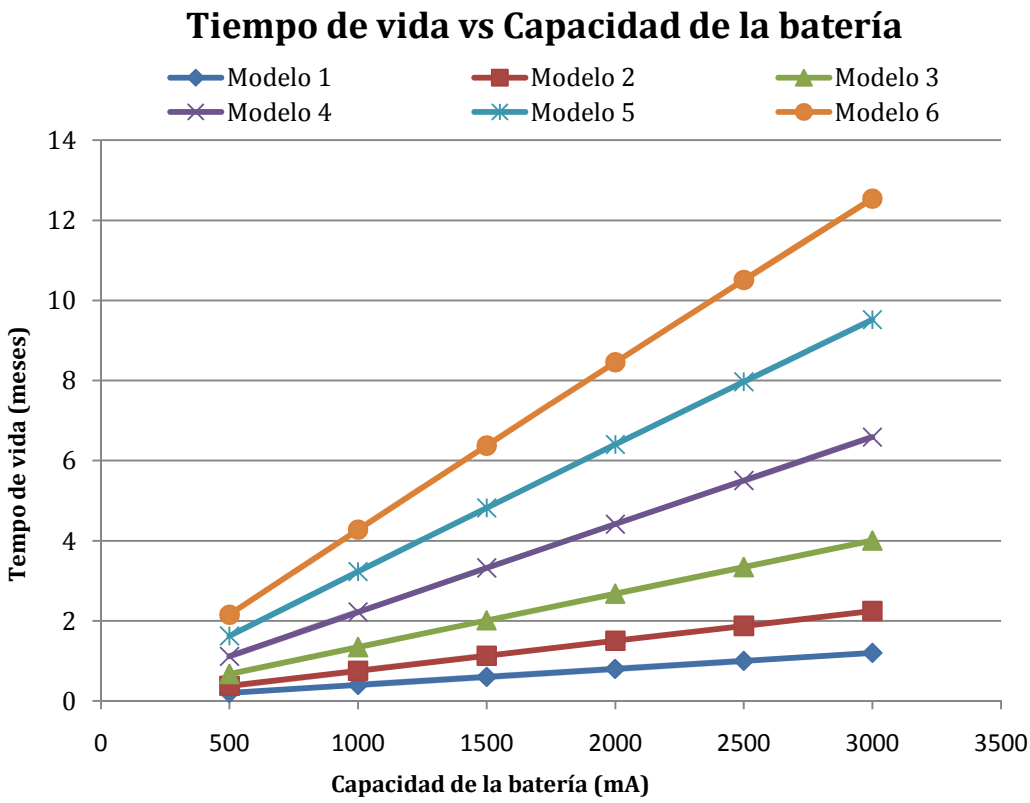


Figura 115: Gráfica duración batería vs capacidad batería (Escenario 3)

La Figura 116 muestra como varia la duración de la batería de 2500 mA-h según el intervalo de monitorización empleada por la aplicación. A diferencia de los anteriores escenarios en este caso el intervalo de monitorización sí que afecta al tiempo de vida de la batería.

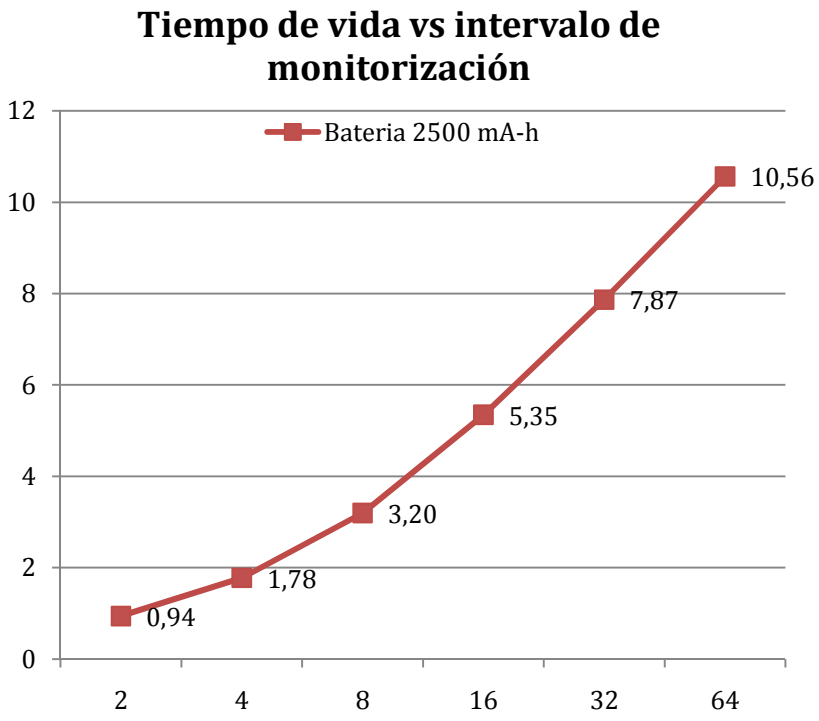


Figura 116: Gráfica tiempo de vida vs intervalo de monitorización (Escenario 3)

8.2 Pruebas en entorno real

Tras comprobar el funcionamiento de la aplicación en un entorno de simulación, es necesario verificar con un entorno real con el despliegue físico de la red que los resultados obtenidos son válidos. En este caso, las pruebas se realizan en el Laboratorio B06, situado en la segunda planta del Edificio Sabatini de la Escuela Politécnica Superior de Leganés. La siguiente figura muestra el plano donde se encuentra el laboratorio.

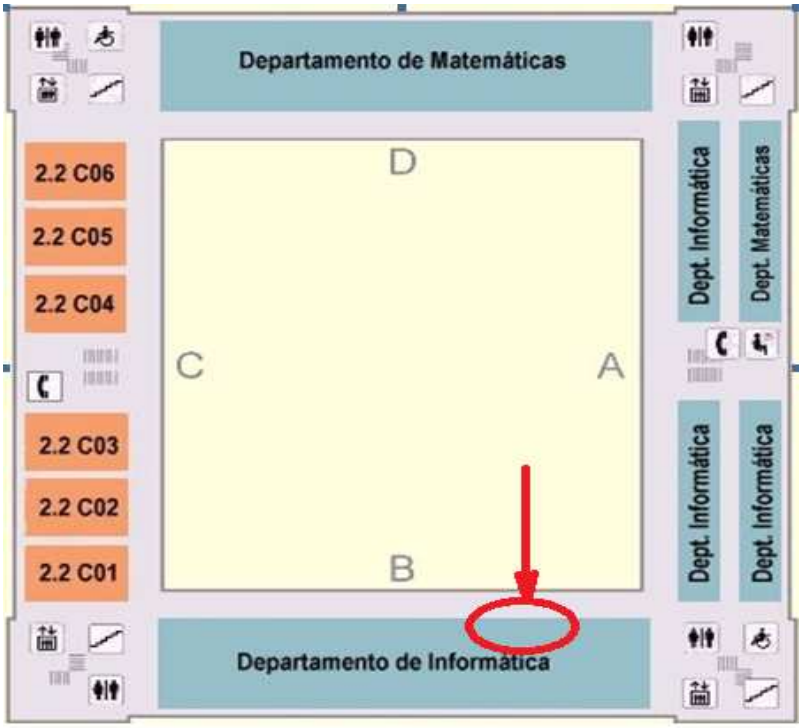


Figura 117: Distribución de la 2ª planta del Edificio Sabatini

Al igual que se hiciera en las pruebas de simulación, se comprobará el buen funcionamiento de los tres escenarios presentados al principio del capítulo (*CTP*, *LQI* y *CTP+LPL*). En el laboratorio se dispone de 3 nodos MicaZ (uno actuará como nodo gateway). El nodo gateway será programado con la aplicación *SensitiveBase*, mientras que los otros dos serán programados con la aplicación a monitorizar (*SensitiveSampler*) usando cada escenario. A continuación se describen los pasos necesarios para la ejecución de una aplicación para WSN.

8.2.1 Compilación de las aplicaciones

La compilación de una aplicación genera un ejecutable que contiene la propia aplicación y el sistema operativo. Para compilar debe situarse en el directorio donde se encuentra la aplicación y ejecutar la siguiente sentencia:

```
$ make platform
```

Donde *platform* indica el modelo de nodo sensor donde la aplicación será instalada.

La Figura 118 muestra un ejemplo de compilación de la aplicación *SensitiveSampler*:

```
Archivo  Editar  Ver  Terminal  Ayuda
Setting up for TinyOS 2.1.0
sueko@sueko-laptop:~$ cd /opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveSampler/
sueko@sueko-laptop:/opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveSampler$ make micaz
mkdir -p build/micaz
    compiling SensitiveSamplerAppC to a micaz binary
ncc -o build/micaz/main.exe  -Os -Wall -Wshadow -Wnesc-all -target=micaz -fnesc-cfile=build/micaz/app.c -board=mts300 -DDEFINED_T
-inline-insns-single=100000 -I.. -I/opt/tinyos-2.1.0/tos/lib/net/ -I/opt/tinyos-2.1.0/tos/lib/net/ctp  -I/opt/tinyos-2.1.0/tos/li
TENING -DIDENT_APPNAME=\"SensitiveSample\" -DIDENT_USERNAME=\"sueko\" -DIDENT_HOSTNAME=\"sueko-laptop\" -DIDENT_USERHASH=0xe0a014
29bbL -DIDENT_UIDHASH=0x96e54bd0L -fnesc-dump=wiring -fnesc-dump='interfaces(!abstract())' -fnesc-dump='referenced(interfacedefs,
e=build/micaz/wiring-check.xml SensitiveSamplerAppC.nc -lm
/opt/tinyos-2.1.0/tos/chips/cc2420/lpl/DefaultLplC.nc:39:2: warning: #warning "*** USING DEFAULT LOW POWER COMMUNICATIONS ***"
    compiled SensitiveSamplerAppC to build/micaz/main.exe
          22834 bytes in ROM
           1586 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe build/micaz/main.ihex
    writing TOS image
sueko@sueko-laptop:/opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveSampler$
```

Figura 118: Proceso de compilación de *SensitiveSampler* para MicaZ

Como se puede observar, el fichero generado se encuentra en el directorio `build/micaz/` de la aplicación y recibe el nombre "`main.exe`". En la Figura 119 muestra el proceso de compilación para la aplicación *SensitiveBase*:

```
Archivo  Editar  Ver  Terminal  Ayuda
Setting up for TinyOS 2.1.0
sueko@sueko-laptop:~$ cd /opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveBase/
sueko@sueko-laptop:/opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveBase$ make micaz
mkdir -p build/micaz
    compiling SensitiveBaseAppC to a micaz binary
ncc -o build/micaz/main.exe  -Os -Wall -Wshadow -Wnesc-all -target=micaz -fnesc-cfile=build/micaz/app.c -board=mts300 -D
-inline-insns-single=100000 -I.. -I/opt/tinyos-2.1.0/tos/lib/net/ -I/opt/tinyos-2.1.0/tos/lib/net/ctp  -I/opt/tinyos-2.1
=\"SensitiveBaseAp\" -DIDENT_USERNAME=\"sueko\" -DIDENT_HOSTNAME=\"sueko-laptop\" -DIDENT_USERHASH=0xe0a01419L -DIDENT_T
0x3f940e6eL -fnesc-dump=wiring -fnesc-dump='interfaces(!abstract())' -fnesc-dump='referenced(interfacedefs, components)'
heck.xml SensitiveBaseAppC.nc -lm
/opt/tinyos-2.1.0/tos/chips/cc2420/lpl/DummyLplC.nc:39:2: warning: #warning "*** LOW POWER COMMUNICATIONS DISABLED ***"
    compiled SensitiveBaseAppC to build/micaz/main.exe
          22438 bytes in ROM
           2223 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe build/micaz/main.ihex
    writing TOS image
sueko@sueko-laptop:/opt/tinyos-2.1.0/apps/SensitiveLPL/SensitiveBase$
```

Figura 119: Proceso de compilación de *SensitiveBase* para MicaZ

Como ocurre con *SensitiveSampler* el fichero ejecutable generado se encuentra en el directorio `build/micaz/` y recibe el nombre "`main.exe`".

Ahora queda la compilación de la aplicación *SensitiveListen*. Se trata de una aplicación Java y está basada en una de las herramientas que vienen en la instalación de TinyOS. El código de la aplicación se encuentra en el directorio `/tinyos-2.1.0/support/sdk/java/net/tinyos/tools` y desde este directorio se compila con el compilador propio de Java (javac):

```
$ javac SensitiveListen.java
```

Este proceso genera un fichero `.class` en el mismo directorio que será movido a la librería `tinyos.jar` que se encuentra en el directorio `/tinyos-2.1.0/support/sdk/java/` para que pueda ser utilizado desde cualquier otro directorio.

Tras la compilación de todas las aplicaciones del sistema, es posible la instalación de las aplicaciones en los nodos.

8.2.2 Instalación de la aplicación en los motes

Se programan los 3 nodos de los que se dispone en el laboratorio, uno con el fichero ejecutable de *SenstitiveBase* y los otros dos con el fichero generado de la compilación de *SensitiveSampler*.

Para el proceso de instalación, ha de utilizarse la placa programadora MIB520. Para la instalación se debe conectar el mote con la placa programadora a través del conector de 51 pines. Posteriormente, se conecta la placa al PC mediante su conector USB y en un terminal se ejecuta el siguiente comando desde el directorio donde se encuentra la aplicación ya compilada:

```
$ make mikaz reinstall,0 mib520,/dev/ttyUSB0
```

Donde `reinstall` instala la aplicación sin volver a compilarla ya que se ha realizado en el paso anterior. Si se quiere compilar la aplicación e instalarla utilice el parámetro `install`. El 0 representa el identificador que se asigna al nodo, téngase en cuenta que la aplicación a instalar en el nodo gateway (*SensitiveBase*) debe llevar el identificador 0 mientras que los nodos sensores (*SensitiveSampler*) llevarán el 1 y el 2 respectivamente. El parámetro `mib520` indica la placa programadora en uso. Por último el parámetro `/dev/ttyUSB0` indica el puerto de programación.

La Figura 120 muestra el proceso de instalación de la aplicación *SensitiveBase* en el nodo gateway.

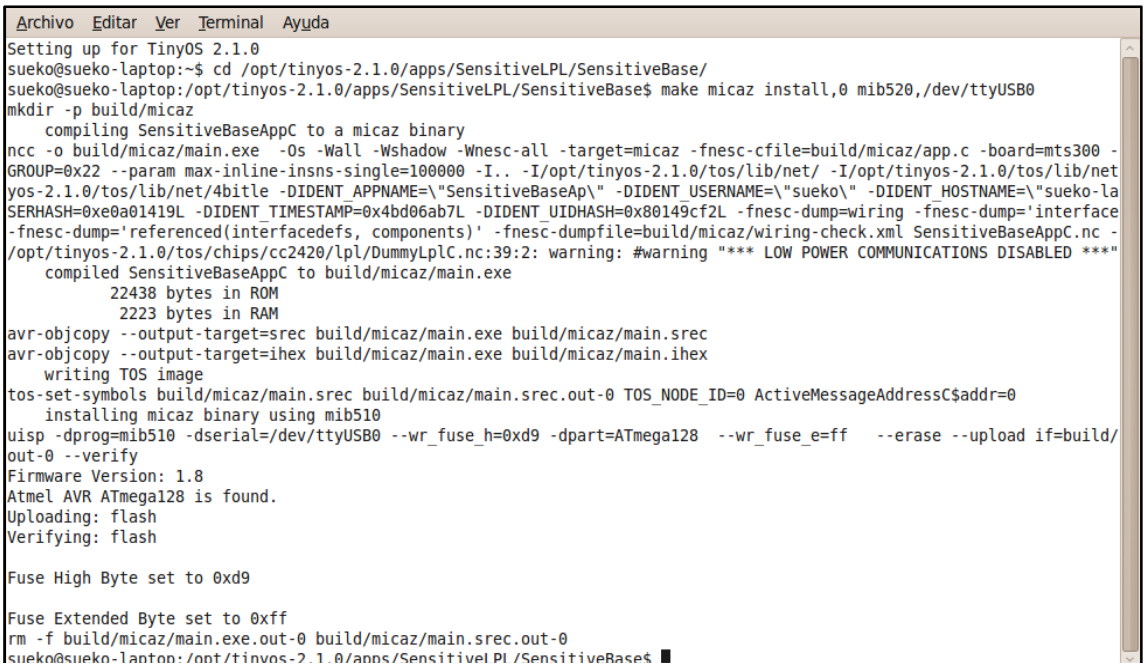


Figura 120: Proceso de instalación de *SensitiveBase* para MicaZ

La Figura 121 muestra el proceso de instalación de la aplicación *SensitiveSampler* en los nodos sensores.

Figura 121: Proceso de instalación de *SensitiveSampler* para MicaZ

Por último se debe mencionar que la aplicación *SensitiveListen* no precisa de instalación en la estación base ya que se utilizará el comando `java` para

interpretar el fichero `SensitiveLiten.class` y lo ejecute como si se tratase de un fichero "`.exe`".

8.2.3 Ejecución de la aplicación

Tras la instalación, es posible la ejecución de las aplicaciones en los nodos. Antes de ejecutar, es necesario comprobar que todos los componentes intervienen en el sistema se encuentren instalados y listos para la ejecución. Estos componentes son:

- Servidor de base de datos MySQL instalado y arrancado.
- Aplicación *SensitiveListen* ejecutando en un terminal de la estación base y escuchando por el puerto para recibir mensajes.
- Nodo gateway preparado y conectado vía USB a la estación base.
- Nodos sensores MicaZ preparados y encendidos.

Para la ejecución de *SensitiveListen* se utilizará el comando java de la siguiente manera:

```
$ java net.tinyos.tools.SensitiveListen –comm serial@/dev/ttyUSB1:micaz
```

Donde el parámetro `–comm serial@/dev/ttyUSB1:micaz` indica el puerto por el que debe escuchar la aplicación para recibir los paquetes de datos.

Una vez iniciada la ejecución de todos los componentes del sistema se debe comprobar su funcionamiento. Para ello en el terminal donde se ejecuta *SensitiveListen* se mostrarán los datos de los paquetes recibidos provenientes del nodo gateway. En la Figura 122 se muestra un ejemplo de la ejecución de *SensitiveListen*.

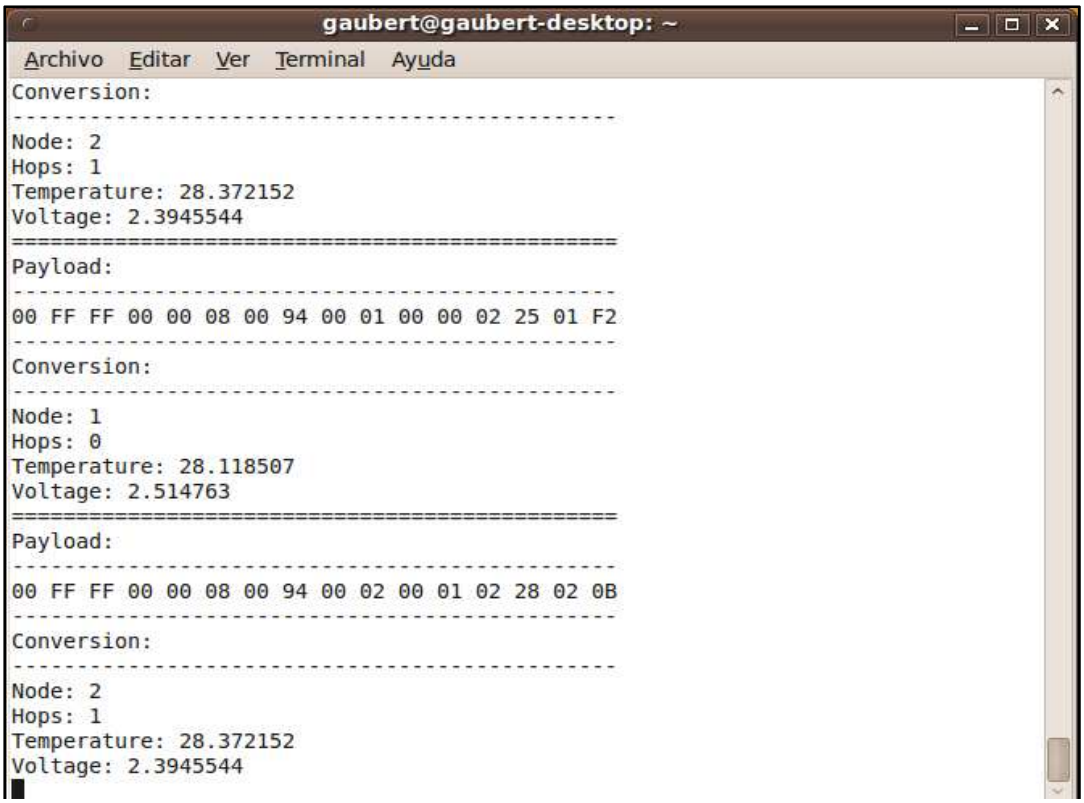


Figura 122: Ejecución de *SensitiveListen* en la estación base

Ahora queda comprobar que la información se está almacenando en la base de datos. Para ello, se utilizará la herramienta *MySQL Browser* que permite la visualización del contenido de las tablas a través de sentencias SQL. Para ello se ejecuta la siguiente sentencia:

```
SELECT * FROM samples;
```

En la Figura 123 se muestra un ejemplo de la información contenida en la base de datos.

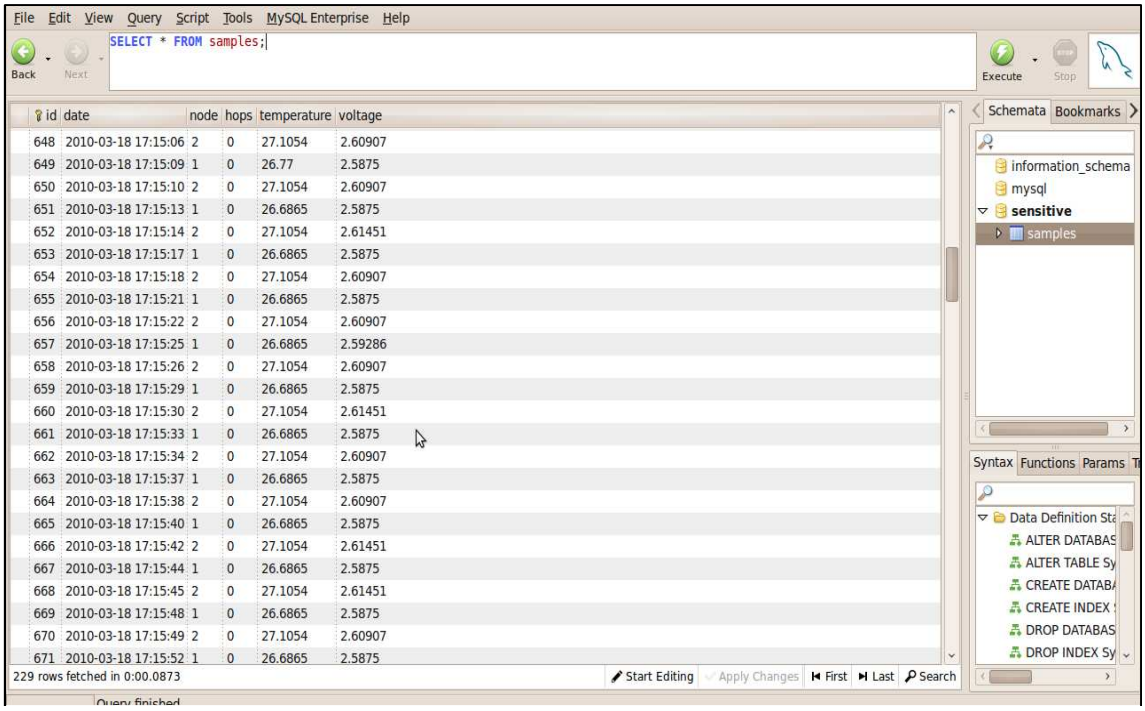


Figura 123: Muestra la información almacenada en la base de datos

8.2.4 Resumen de los datos obtenidos

En este apartado se estudian los datos recibidos por los nodos sensores y que se encuentran almacenados en la base de datos. A continuación se estudian los datos referentes a las lecturas del sensor de temperatura y del voltaje de la batería del nodo.

8.2.4.1 Temperatura

Se estudiarán los datos de temperatura recibidos de los nodos sensores durante su ejecución y comprobar la coherencia de los mismos. Teniendo en cuenta que la sala donde se han realizado las pruebas es un laboratorio del departamento de informática, con varios computadores y servidores funcionando durante las 24 horas del día, por lo que es de esperar que la temperatura sea elevada a pesar de contar con climatización en el laboratorio.

La Figura 124 muestra la temperatura registrada por los sensores durante su ejecución. El comportamiento del sensor es el se esperaba, pues los nodos se encuentran cerca de los ordenadores que actúan como fuentes de calor.

Ahora queda comprobar que ambos sensores de temperatura estén tomando datos fiables. Se ha comprobado que el sensor de temperatura de uno de los nodos parece estar tomando valores correctos. Queda por comprobar los datos del otro nodo y cotejar los valores de ambos para comprobar que toman valores coherentes. En la Figura 124 se puede observar como ambos motes están tomando valores muy similares, con una diferencia máxima de ± 0.3ºC que para el objetivo de este proyecto es más que suficiente.

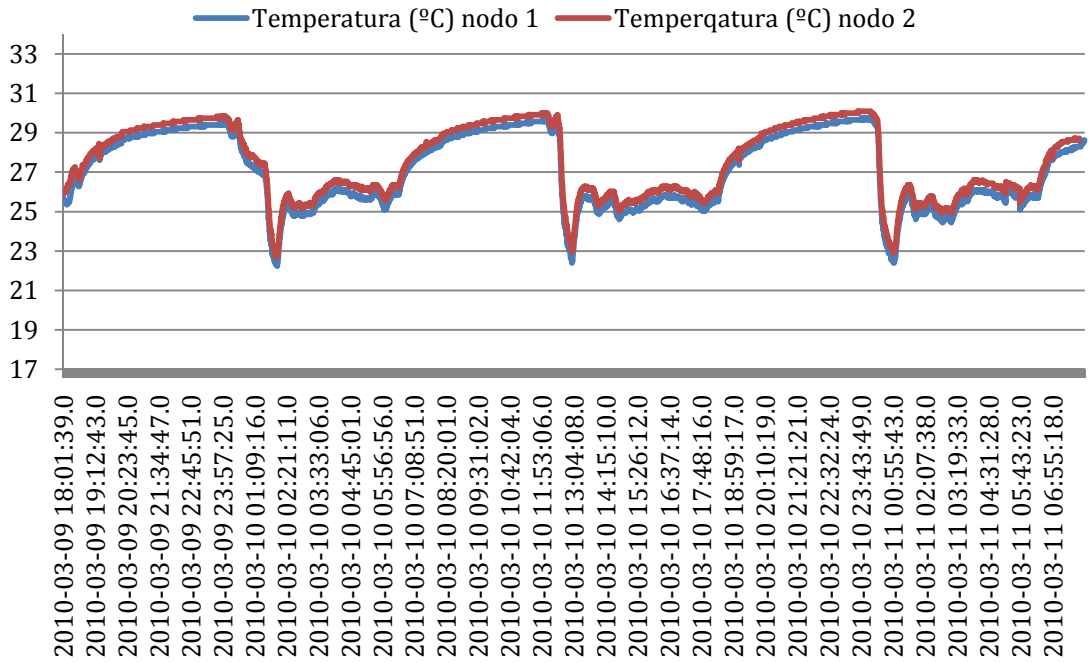


Figura 124: Temperatura registrada por dos sensores de forma paralela

En la Figura 124 puede observarse que el comportamiento de la temperatura en el laboratorio repite un patrón diario, alcanzando una temperatura máxima de 29.4 °C y una temperatura mínima de 22.1 °C. Se podría afirmar que el laboratorio se encuentra expuesto a temperaturas que están por encima de lo que parece razonable para un lugar de trabajo. Como se ha mencionado con anterioridad, la razón de estas temperaturas tan elevadas se debe al calor que desprende las decenas de PCs que se encuentran encendidos 24 horas al día. Por tanto, el laboratorio no parece mantener un comportamiento eficiente del consumo de los sistemas de calefacción y refrigeración, obtendría pues, una mala

calificación en el certificado energético relativo a la Directiva 2002/91/CE del parlamento europeo.

8.2.4.2 Voltaje

Queda por comprobar los datos del voltaje de las baterías. Para ello se analizará el comportamiento del voltaje de las baterías para los tres escenarios vistos en apartados anteriores:

- Escenario 1: *SensitiveSampler* con CTP.
- Escenario 2: *SensitiveSampler* con LQI.
- Escenario 3: *SensitiveSampler* con CTP+LPL.

En la Figura 125 se observa que la tendencia del consumo de las baterías en el nodo ejecutando *SensitiveSampler* con CTP y un intervalo de monitorización de 32 segundos. La batería consta de dos pilas recargables Panasonic modelo HHR-3MRE (Tipo AA), con una capacidad de 2100 mAh y un voltaje de 1.2 voltios cada una.

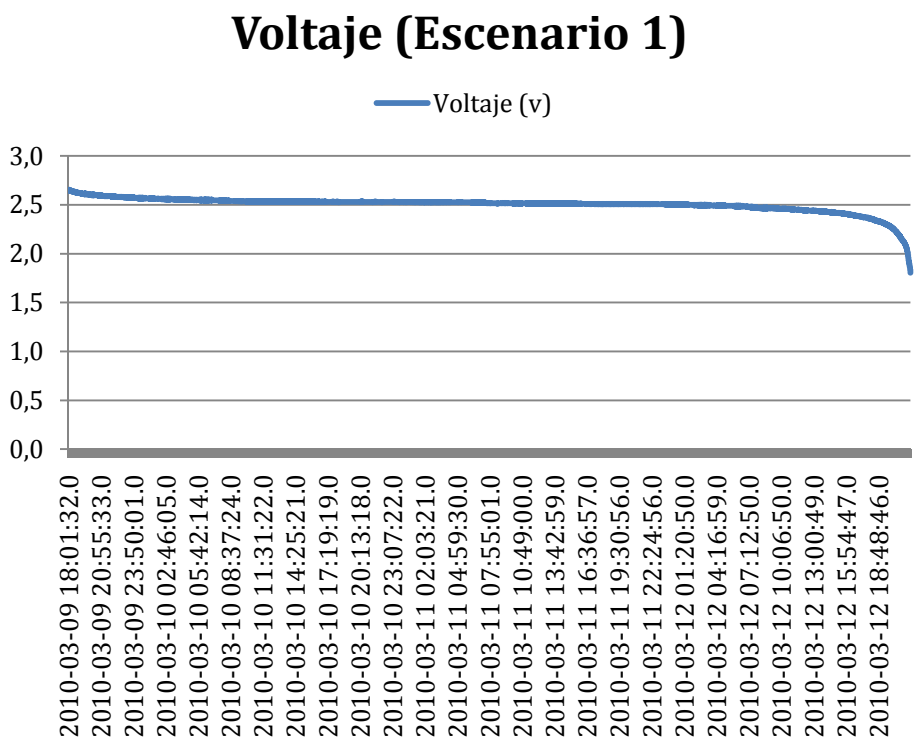


Figura 125: Comportamiento de la batería (Escenario 1)

En la Figura 126 se observa la tendencia del consumo de las baterías en el nodo ejecutando *SensitiveSampler* con LQI y un intervalo de monitorización de 32 segundos. La batería, como en el escenario anterior, consta de dos pilas recargables. Cabe mencionar que, las baterías utilizadas para las pruebas en entorno real tienen un voltaje de 1.2 voltios, mientras que en las simulaciones Avrora estipula que las baterías son de 1.5 voltios. Esta es la causa de la diferencia de duración de las baterías en entorno real y en simulación, de 3 días y 2 horas en ejecución real mientras que el cálculo en simulación es de 5 días y medio.

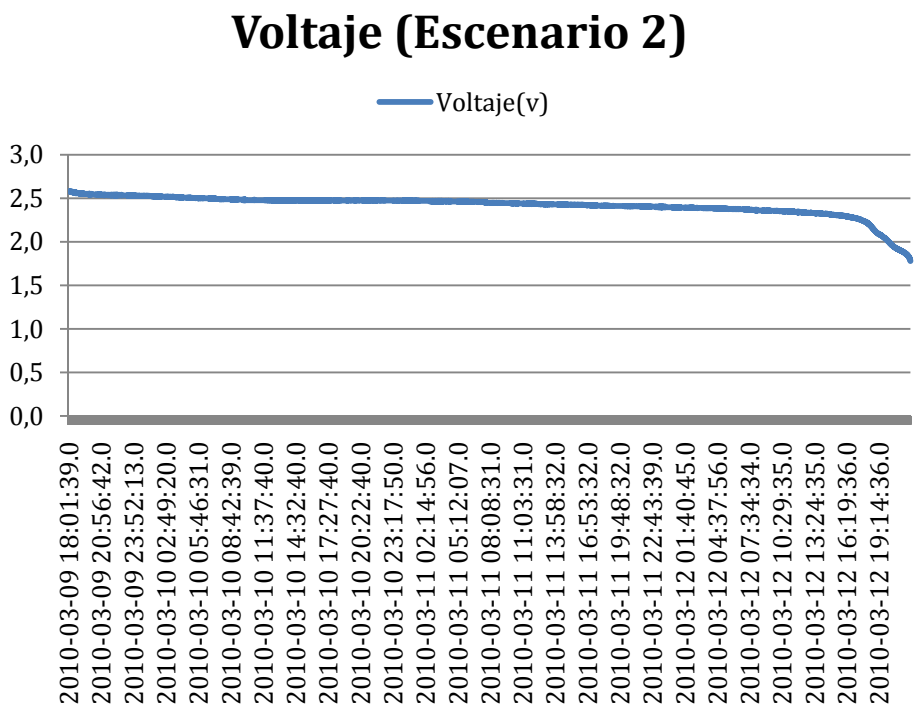


Figura 126: Comportamiento de la batería (Escenario 2)

En la Figura 127 se muestra la tendencia del consumo de las baterías en el nodo ejecutando *SensitiveSampler* con CTP + LPL y un intervalo de monitorización de 4 segundos. Las baterías utilizadas fueron las mismas que en los escenarios anteriores. Se observa que el comportamiento del voltaje es distinto al de los anteriores escenarios. La duración de las baterías parece tener coherencia con los resultados obtenidos. Ya que, en simulación daba como resultado una duración de 3.34 meses y en la gráfica se observa que el voltaje ha bajado apenas 0.2 voltios en más de 20 días de ejecución.

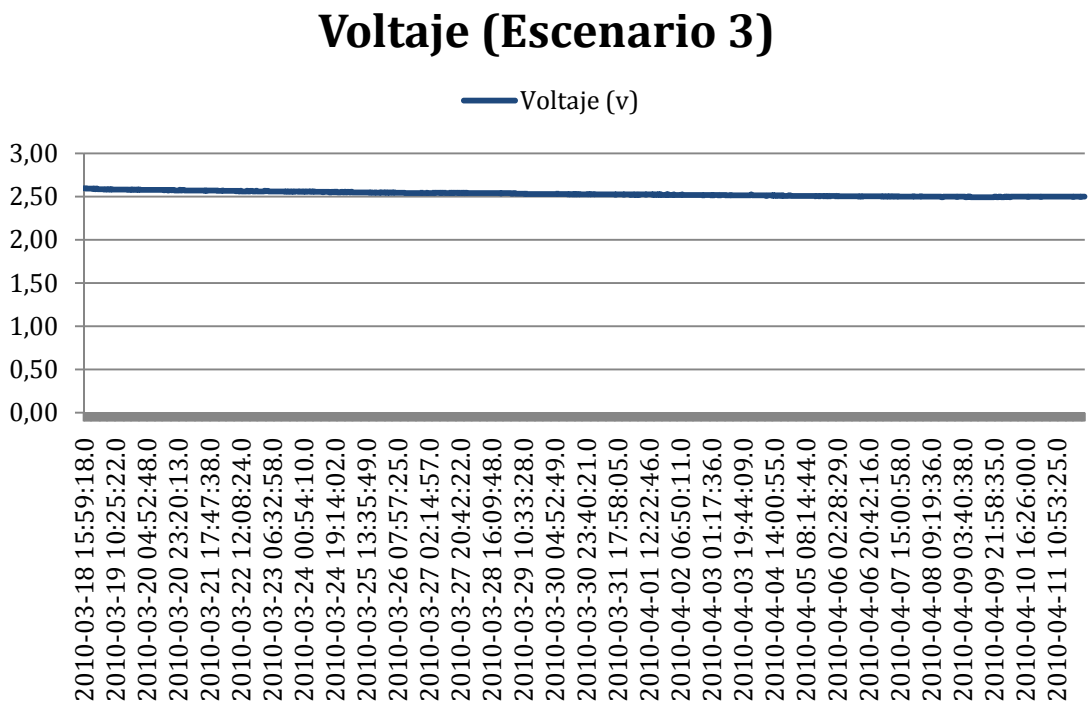


Figura 127: Comportamiento de la batería (Escenario 3)

Puede observarse el comportamiento del voltaje de las baterías entre los distintos escenarios. En los escenarios 1 y 2 el comportamiento es similar, y como predecían las simulaciones tienen una duración de pocos días en ambos casos. En el escenario 3 el comportamiento es distinto debido al uso de políticas de ahorro de energía como *Low Power Listening* que proporciona un ahorro de energía importante y la batería tiene una duración mucho más longeva.

Capítulo 9 Conclusiones

El presente capítulo se centra en la revisión de los objetivos del proyecto marcados en el Capítulo 1, así como en la evaluación de los mismos. Además, se proponen líneas futuras de trabajo e investigación para favorecer el desarrollo de otros proyectos.

9.1 Revisión de los objetivos

Se tomará como punto de partida un análisis sobre los objetivos propuestos en el Capítulo 1 y se cotejarán con lo logrado a lo largo del proyecto.

EL principal objetivo de este proyecto era construir una aplicación para redes de sensores que monitorice la temperatura ambiente de un edificio. Con el objetivo de determinar la eficiencia energética del edificio sometido a estudio, según lo estipulado en la Directiva 2002/91/CE del Parlamento Europeo. Al localizar los nodos en un área específica del edificio, no es posible cubrir toda el área deseada, pero será suficiente para dar una idea general sobre la eficiencia energética del edificio.

Otro de los principales objetivos del presente proyecto es la optimización de energía de la propia red de sensores. Para obtener un estudio fiable sobre la eficiencia energética de un edificio, es necesario que la red de sensores tenga un tiempo de vida suficiente para cometer dicho objetivo. Por ello, controlar el consumo energético de un nodo sensor es uno de los principales escoyos que se han salvado durante la realización del proyecto. Como se ha demostrado en el Capítulo 8 la red de sensores puede llegar a tener una vida útil de más de un año de duración.

Un objetivo importante que se ha conseguido, es que ha sido posible determinar el tiempo de vida estimado de las baterías y, en consecuencia, el tiempo de vida de la red. Este es un aspecto fundamental en el desarrollo de aplicaciones de redes de sensores, sobre todo, en aquellas aplicaciones que se

ejecuten en entornos donde una vez desplegada, resulte imposible acceder para el cambio de las baterías.

Minimizar del coste del proyecto es otro de los objetivos del presente proyecto. El coste del proyecto se puede dividir en dos partes (hardware y software). Respecto al hardware, el precio de cada nodo sensor es fijo y de ahí no es posible reducir el coste. Pero por parte del software al utilizar sistemas con licencias GPL (TinyOS, Java, MySQL, Ubuntu, etc.) se ha reducido al mínimo su coste (coste 0).

Obtener un sistema flexible, escalable y tolerante a fallos era otro de los objetivos del proyecto. Con la utilización del sistema operativo para nodos *TinyOS* y de algoritmos de enrutamiento como *Collection Tree Protocol* que permiten la consecución de estos objetivos.

9.2 Líneas futuras de trabajo

A lo largo del desarrollo de este proyecto, han surgido algunas posibilidades de mejora. A continuación se exponen aquellas que pueden ser objeto de estudio y de trabajos futuros:

La seguridad en las comunicaciones de la red de sensores inalámbricos es uno de los aspectos más importantes para el desarrollo de las WSN. La implementación de los actuales algoritmos de cifrado resulta impensable por el coste energético que supone. Actualmente existen proyectos para implementar algoritmos ligeros de cifrado para dotar de seguridad a las comunicaciones de la red de sensores.

Sería interesante dotar a la red de comunicación inversa, es decir, desarrollar funcionalidades que permitan configurar los nodos sensores (cambiar el intervalo de monitorización, solicitar información en tiempo real de algún nodo sensor, etc.). Existen implementaciones de protocolos que permiten este tipo de comunicación. Uno de los más utilizados en TinyOS es *Dissemination*.

Otra de las posibles líneas de futuro seria la inclusión de actuadores en este tipo de redes de sensores que permitan realizar acciones sobre dispositivos como sistemas de climatización, cámaras, rotores de persianas, etc.

Además, resultaría interesante la evaluación de los nuevos protocolos de enrutamiento desarrollados para las redes de sensores inalámbricas en términos de consumo energético. Por ejemplo el estudio del protocolo *6lowpan* que dota a los sensores de direcciones IPv6 (42) determinado el coste energético que supone la utilización de dicho protocolo en los nodos sensores.

Por último, está el campo de estudio de la geolocalización de sensores. Sería interesante disponer de métodos que permitan conocer la localización exacta sobre un plano del sensor. En la actualidad existen algoritmos como RSSI que dan aproximaciones de la distancia del nodo a ciertos nodos "baliza" colocados estratégicamente. Este es uno de los campos en el que últimamente la comunidad de usuarios de TinyOS está trabajando más intensamente.

9.3 Presupuesto

Para la realización del presupuesto necesario para la elaboración de este proyecto es necesario llevar a cabo un análisis de costes de los recursos utilizados que se dividirán en recursos materiales y recursos humanos.

9.3.1 Recursos humanos

En el caso de los costes de recursos humanos, se tienen en cuenta cada una de las fases que conforman el proyecto. Aplicando el *Modelo en Cascada* ciclo de vida de una aplicación software, se tienen las siguientes fases:

- Análisis del entorno del proyecto y adquisición de conocimientos sobre el estado de la cuestión.
- Análisis del problema.
- Diseño.
- Implementación.
- Evaluación y pruebas.

- Documentación.

El coste humano (esfuerzo) se establece teniendo en cuenta las siguientes consideraciones:

- El proyecto ha sido desarrollado por un único recurso (persona), que trabajará a jornada completa de lunes a viernes, haciendo un total de 40 horas semanales.
- La fecha de comienzo del proyecto ha sido el 1 de Diciembre del 2009 y la fecha de finalización el 30 Abril de 2010, lo que hace un total de 101 días trabajados que en horas trabajadas son 808.

Para determinar el coste en términos monetarios de los recursos humanos implicados en el proyecto, se considera una retribución de 30€/hora. La Tabla 10 muestra la distribución de las horas empleadas en la consecución de cada una de las fases en las que se divide el proyecto:

Tabla 10: Costes de recursos humanos

	Precio/hora (€)	Horas	Coste de la fase (€)
Análisis del entorno	30	80	2.400
Análisis del problema	30	80	2.400
Diseño	30	160	4.800
Implementación	30	120	3.600
Evaluación y pruebas	30	208	6.240
Documentación	30	160	4.800
Total	30	808	24.240

9.3.2 Recursos materiales

Para la evaluación de los costes materiales, se tendrán en cuenta todos los elementos hardware necesarios para la implementación del proyecto final, desde el PC donde se desarrolla y se simula, hasta los motes físicos para la evaluación real.

Todo proyecto software tiene asociados unos costes de materiales y servicios, además de los descritos en el punto anterior. La Tabla 11 muestra el coste unitario y total de cada uno de los elementos necesarios:

Tabla 11: Recursos materiales utilizados en este proyecto

	Cantidad	Coste (€)	Subtotal (€)
Ordenador portátil	1	700	700
Conexión a internet (coste mensual)	5	40	200
Impresión del proyecto (en tapa dura)	2	60	120
Impresión del proyecto (en espiral)	3	35	105
Placa sensora MTS310CB	2	190,156	380,312
Mote Micaz (MPR2400CA)	3	100	300
Placa programadora MIB520	1	74,55	74, 55
Pilas recargables Panasonic hhr-3mre	4	3,74	14,95
Total			1.894,812

Los precios de la placa sensora, las motas y la placa programadora se han obtenido de la página oficial de productos de Crossbow (43).

9.3.3 Costes totales

En la Tabla 12 se muestra un resumen con el coste total del proyecto:

Tabla 12: Costes totales del proyecto

	Coste (€)
Recursos humanos	24.240
Recursos materiales	1.894,812
Total	26.134,812

9.4 Evaluación personal

En este capítulo de conclusiones se han repasado los objetivos del proyecto que se han alcanzado y los que se pretendían conseguir y se han indicado las posibles mejoras que se pueden realizar sobre el trabajo desarrollado, con el objetivo de obtener una solución más completa.

Desde el punto de vista académico, destaca la intensa labor de investigación desarrollada y el esfuerzo que esto ha supuesto. Se han estudiados tecnologías completamente nuevas: conocer la arquitectura de un sistema operativo como *TinyOS*, programar este tipo de dispositivos embebidos, conocer las particularidades de un lenguaje de programación orientado a componentes como *nesC* y el empleo de un estándar de comunicación como *ZigBee*. Antes de la

elaboración de este proyecto todas las tecnologías empleadas eran totalmente desconocidas.

En el aspecto personal, durante el tiempo dedicado al desarrollo de este proyecto he reafirmado la decisión que tomé, hace tiempo, de estudiar una carrera de Informática. Que tras mi paso por el mundo laboral en empresas de tecnologías de la información (TICs) había llegado a poner en duda. El estudio de una nueva tecnología, que tiene tantos campos de abiertos de investigación, con tantas posibilidades ha resultado, por mi parte, emocionante y muy satisfactorio.

Bibliografía

1. Directiva 2002/91/CE del Parlamento Europeo y del Consejo. [En línea] http://europa.eu/legislation_summaries/energy/energy_efficiency/l27042_es.htm

2. Sun SPOT World. [En línea] http://www.sunspotworld.com/

3. IEEE 802.15 WPAN™ Task Group 4 (TG4). [En línea] http://www.ieee802.org/15/pub/TG4.html

4. ZigBee Alliance. [En línea] http://www.zigbee.org/

5. Crossbow Technology Inc. [En línea] http://www.xbow.com/

6. Infineon Technologies. [En línea] http://www.infineon.com/cms/en/product/index.html

7. Texas Instruments. [En línea] http://www.ti.com/

8. Shockfish SA. [En línea] http://www.tinynode.com/

9. CodeBlue: Wireless Sensors for Medical Care. [En línea] http://fiji.eecs.harvard.edu/CodeBlue

10. TinyOS Community Forum. [En línea] http://www.tinyos.net/

11. Contiki. The Operating System for Embedded Smart Objects. [En línea] http://www.sics.se/contiki/

12. MANTIS OS: An Embedded Multithreaded Operating. [En línea] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.9081&rep=rep1&type=pdf

13. LiteOS. [En línea] http://www.liteos.net/

14. Bertha, Pushpin Computing System Overview. [En línea]
http://www.media.mit.edu/resenv/pubs/papers/2002-09-PushpinPervasiveWF.pdf

15. CORMOS: A Communication-Oriented Runtime System for Sensor Networks. [En línea] http://www.ics.forth.gr/~bilas/pdffiles/cormos-ewsn05-cr.pdf

16. eCos, embedded Configurable operating system. [En línea]
http://ecos.sourceware.org/

17. MagnetOS OPERATING SYSTEM. [En línea]
http://www.cs.cornell.edu/People/egs/magnetos/

18. NutOS Embedded Ethernet Devices. [En línea]
http://www.proconx.com/xnut/nutos/

19. t-kernel: Providing Reliable OS Support to Wireless Sensor. [En línea]
http://www.cs.virginia.edu/~stankovic/psfiles/fp01-gu.pdf

20. nesC: A Programming Language for Deeply Networked Systems. [En línea]
http://nescc.sourceforge.net/

21. The MicaZ Wireless Platform for Low-Power Sensor Networks. [En línea]
http://www.xbow.com/products/productdetails.aspx?sid=164

22. MicaZ Datasheet . [En línea]
http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf

23. MTS300CB Datasheet. [En línea]
http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MTS_MDA_Datasheet.pdf.

24. MIB520 Datasheet. [En línea]
http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB520_Datasheet.pdf

25. TinyOS Programming. [En línea] http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf

26. Gateway Middleware Xserve. [En línea] http://www.xbow.com/Technology/GatewayMiddleware.aspx

27. WSN Professional Kit. [En línea] http://www.xbow.com/Products/productdetails.aspx?sid=231

28. Java Sun Microsystems. [En línea] http://java.com/es/

29. Licencia Pública General. [En línea] http://www.gnu.org/licenses/licenses.es.html

30. System Architecture for Wireless Sensor Networks. [En línea] http://www.jlhlabs.com/jhill_cs/jhill_thesis.pdf

31. Oracle. [En línea] http://www.oracle.com/index.html

32. AVRORA The AVR Simulation and Analysis Framework. [En línea] http://compilers.cs.ucla.edu/avrora/

33. Canonical Ltd. [En línea] http://www.ubuntu.com/

34. MySQL Community Server. [En línea] http://dev.mysql.com/downloads/mysql/

35. The Unified Modeling Language. [En línea] http://www.uml.org/

36. The Link Estimation Exchange Protocol (LEEP). [En línea] http://www.tinyos.net/tinyos-2.x/doc/html/tep124.html

37. The Collection Tree Protocol (CTP). [En línea] http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html

38. Energy measurements for MicaZ node. [En línea] http://vs.cs.uni-kl.de/en/publications/2006/KrGe06/energy.pdf

39. Accurate prediction of power consumption in sensor networks. [En línea] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.6036&rep=rep1&type=pdf

40. A Review of Current Operating Systems for Wireless Sensor. [En línea] http://www.comp.nus.edu.sg/~doddaven/cata.pdf

41. **Rubén García Olalla.** Monitorización de la temperatura de un edificio mediante una red de sensores inalámbrica usando motes MicaZ. [En línea] http://e-archivo.uc3m.es/bitstream/10016/6138/1/PFC_Ruben_Garcia_Olalla.pdf

42. IPv6. [En línea] http://www.ipv6.org/

43. Crossbow's Products Overview. [En línea] http://www.xbow.com/Products/wproductsoverview.aspx

Anexo A Avrora

A.1 Instalación

La instalación de Avrora es un proceso sencillo. En este anexo se describen los pasos necesarios para su obtención e instalación para el sistema operativo Ubuntu en su versión 9.10 "Karmic Koala".

A.1.1 Paso 1

Avrora está implementado en Java, y por lo tanto requiere de una máquina virtual de Java para su funcionamiento. Será preciso tener instalado previamente Java 4 o una versión más reciente. Una vez que instalada la JVM es posible ejecutar aplicaciones Java en la línea de comandos desde un terminal. Para obtener Java de *Sun Microsystems* ir a: http://java.sun.com.

A.1.2 Paso 2

La plataforma Java permite integrar aplicaciones en archivos comprimidos ".jar", con el objetivo de facilitar su instalación y ejecución. De esta forma las aplicaciones no requieren de compilación, instalación o configuración previa y pueden ejecutarse directamente desde el archivo ".jar". Para obtener y probar Avrora tan solo es necesario descargarse la última versión del archivo JAR (1.7.106) en: http://compilers.cs.ucla.edu/avrora/jars/ y almacenarlo en disco para su posterior uso.

A.1.3 Paso 3

Una vez almacenado en disco el archivo ".jar" es necesario actualizar la variable CLASSPATH para poder ejecutar Avrora desde cualquier directorio.

```
$ export CLASSPATH=AVRORAROOT/avrora-beta-1.7.106.jar;$CLASSPATH
```

Donde:

- AVRORAROOT: Es la ruta donde se encuentra almacenado el archivo ".jar".

A.2 Opciones de simulación

Para la simulación de aplicaciones se asume que previamente se ha instalado TinyOS 2.x. A continuación se muestran los pasos a seguir para realizar una simulación con Avrora:

```
$ cd $TOSROOT/apps/Blink
$ make micaz
$ mv build/micaz/main.exe Blink.elf
$ java avrora.Main -platform=micaz -simulation=sensor-network -
seconds=3 -monitors=leds Blink.elf
```

Antes de realizar la simulación Avrora precisa del ejecutable de la aplicación tras la compilación de un programa. Además es necesario que el fichero de entrada para la simulación sea un ".elf". Tras esto, es posible realizar la simulación, que se lleva cado a través del comando `java avrora.Main`. También es posible configurar la simulación según las necesidades a través de los parámetros de entrada. A continuación se describen los de mayor uso:

- `platform`: indica la plataforma en la que están construidos los nodos sensores, admite dos valores: `mica2` y `micaz`.
- `simulation`: tipo de simulación a realizar. Admite dos valores:
  - `sensor-network`: para simula una red de sensores.
  - `single`: simula un solo nodo ejecutando una aplicación.
- `nodecount`: indica el número de nodos que ejecutan las aplicaciones.
- `seconds`: tiempo de la simulación.
- `monitors`: indica las diferentes variables a monitorizar durante la simulación. De entre todas la posibilidades destaca:
  - `packet`: muestra en crudo los paquetes enviados y recibidos por los nodos de la red de sensores.
  - `energy`: muestra un análisis sobre el consumo de cada nodo, y de cada componente del nodo.

- `calls`: muestra la traza de llamadas y retornos a código que se producen durante la simulación.
- `memory`: muestra información y estadísticas sobre la memoria utilizada por la aplicación durante la simulación.
- `leds`: muestra el comportamiento (on/off) de cada uno de los LEDs durante la simulación.

Para obtener mayor información y ayuda sobre las posibilidades de Avrora visite la web: http://compilers.cs.ucla.edu/avrora/help/help.html.

Anexo B Cálculos de los modelos energéticos

B.1 Sensitive con CTP

Tabla 13: Cálculos del modelo energético de Sensitive con CTP

		Modelo 1 2 seg	Modelo 2 4 seg	Modelo 3 8 seg	Modelo 4 16 seg	Modelo 5 32 seg	Modelo 6 64 seg
Microprocesador	(A)	(%)	(%)	(%)	(%)	(%)	(%)
Active	0,0075667	0,2115	0,1334	0,0941	0,0745	0,0646	0,0598
Idle	0,0033433	99,7885	99,8666	99,9059	99,9255	99,9354	99,9402
Noise Reduction	0,0009884	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Power Down	0,0001158	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Power Save	0,0001237	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Reserved 1	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Reserved 2	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Standby	0,0002356	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Extended Standby	0,0002433	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Radio							
Power Off	0,00000002	0,0302	0,0302	0,0302	0,0302	0,0302	0,0302
Power Down	0,0000200	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Idle	0,0004260	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Receive (Rx)	0,0188000	99,9047	99,9369	99,9530	99,9611	99,9652	99,9672
Transmit (Tx)	0,0174000	0,0651	0,0329	0,0168	0,0087	0,0046	0,0026
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Red							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Yellow							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Green							
On	0,0022000	11,0052	5,5496	2,7541	1,2110	0,6816	0,3363
Off	0,0000000	88,9948	94,4504	97,2459	98,7890	99,3184	99,6637
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Flash Memory							
Standby	0,0000020	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Write	0,0040000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Read	0,0150000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Load	0,0000020	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Sensor Board							
On	0,0000700	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Off	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
A-hr consumidos		**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**
uP		0,00335223	0,00334893	0,00334727	0,00334645	0,00334603	0,00334583
Radio		0,01879341	0,01879386	0,01879409	0,01879420	0,01879426	0,01879429
Flash Memory		0,00000200	0,00000200	0,00000200	0,00000200	0,00000200	0,00000200
Leds		0,00024211	0,00012209	0,00006059	0,00002664	0,00001500	0,00000740
Sensor Board		0,00007000	0,00007000	0,00007000	0,00007000	0,00007000	0,00007000
Total		**0,022460**	**0,022337**	**0,022274**	**0,022239**	**0,022227**	**0,022220**

B.2 Sensitive con LQI

Tabla 14: Cálculos del modelo energético de Sensitive con LQI

		Modelo 1 2 seg	Modelo 2 4 seg	Modelo 3 8 seg	Modelo 4 16 seg	Modelo 5 32 seg	Modelo 6 64 seg
Microprocesador	(A)	(%)	(%)	(%)	(%)	(%)	(%)
Active	0,0075667	0,1780	0,1179	0,0973	0,0722	0,0647	0,0609
Idle	0,0033433	99,8220	99,8821	99,9027	99,9278	99,9353	99,9391
Noise Reduction	0,0009884	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Power Down	0,0001158	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Power Save	0,0001237	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Reserved 1	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Reserved 2	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Standby	0,0002356	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Extended Standby	0,0002433	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Radio							
Power Off	0,00000002	0,0302	0,0302	0,0302	0,0302	0,0302	0,0302
Power Down	0,0000200	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Idle	0,0004260	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Receive (Rx)	0,0188000	99,9000	99,9331	99,9497	99,9579	99,9621	99,9641
Transmit (Tx)	0,0174000	0,0698	0,0367	0,0201	0,0119	0,0077	0,0057
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Red							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100	100,0000	100,0000	100,0000	100,0000	100,0000
LED Yellow							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100	100,0000	100,0000	100,0000	100,0000	100,0000
LED Green							
On	0,0022000	10,8864	5,4370	2,7116	1,3499	0,6749	0,3306
Off	0,0000000	89,1136	94,5630	97,2884	98,6501	99,3251	99,6694
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Flash Memory							
Standby	0,0000020	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Write	0,0040000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Read	0,0150000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Load	0,0000020	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Sensor Board							
On	0,0000700	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Off	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
A-hr consumidos		**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**
uP		0,00335082	0,00334828	0,00334741	0,00334635	0,00334603	0,00334587
Radio		0,01879335	0,01879381	0,01879404	0,01879416	0,01879421	0,01879424
Flash Memory		0,00000200	0,00000200	0,00000200	0,00000200	0,00000200	0,00000200
Leds		0,00023950	0,00011961	0,00005966	0,00002970	0,00001485	0,00000727
Sensor Board		0,00007000	0,00007000	0,00007000	0,00007000	0,00007000	0,00007000
Total		**0,022456**	**0,022334**	**0,022273**	**0,022242**	**0,022227**	**0,022219**

B.3 Sensitive con CTP y LPL

Tabla 15: Cálculos del modelo energético de Sensitive con CTP y LPL

		Modelo 1 2 seg	Modelo 2 4 seg	Modelo 3 8 seg	Modelo 4 16 seg	Modelo 5 32 seg	Modelo 6 64 seg
Microprocesador	(A)	(%)	(%)	(%)	(%)	(%)	(%)
Active	0,0075667	0,5155	0,2028	0,1285	0,0955	0,0778	0,0688
Idle	0,0033433	14,3667	7,2452	3,6552	1,8504	0,9880	0,5218
Noise Reduction	0,0009884	0,0005	0,0024	0,0012	0,0006	0,0003	0,0002
Power Down	0,0001158	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Power Save	0,0001237	84,6274	92,2896	96,0781	97,9788	98,8900	99,3793
Reserved 1	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Reserved 2	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Standby	0,0002356	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Extended Standby	0,0002433	0,4899	0,2600	0,1370	0,0747	0,0439	0,0299
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Radio							
Power Off	0,00000002	85,3997	92,6403	96,2873	98,1173	98,9936	99,4668
Power Down	0,0000200	0,0939	0,0475	0,0241	0,0126	0,0068	0,0039
Idle	0,0004260	0,0096	0,0050	0,0025	0,0012	0,0007	0,0004
Receive (Rx)	0,0188000	14,4319	7,2744	3,6695	1,8603	0,9944	0,5264
Transmit (Tx)	0,0174000	0,0649	0,0328	0,0166	0,0086	0,0045	0,0025
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Red							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Yellow							
On	0,0022000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Off	0,0000000	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
LED Green							
On	0,0022000	9,8561	4,9973	2,5134	1,2110	0,6398	0,3226
Off	0,0000000	90,1439	95,0027	97,4866	98,7890	99,3602	99,6774
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Flash Memory							
Standby	0,0000020	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Write	0,0040000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Read	0,0150000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
Load	0,0000020	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Sensor Board							
On	0,0000700	100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
Off	0,0000000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
		100,0000	100,0000	100,0000	100,0000	100,0000	100,0000
A-hr consumidos		**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**	**(A-hr)**
uP		0,00062521	0,00037239	0,00025112	0,00019048	0,00016136	0,00014566
Radio		0,00272457	0,00137334	0,00069279	0,00035126	0,00018775	0,00009942
Flash Memory		0,00000200	0,00000200	0,00000200	0,00000200	0,00000200	0,00000200
Leds		0,00021683	0,00010994	0,00005529	0,00002664	0,00001408	0,00000710
Sensor Board		0,00007000	0,00007000	0,00007000	0,00007000	0,00007000	0,00007000
Total		**0,003639**	**0,001928**	**0,001071**	**0,000640**	**0,000435**	**0,000324**